Janis

Michael Franklin, Richard Lupat

Nov 19, 2021

Contents

1 Introduction 3

2 Toolbox 5

3 Support 7

4 Contents 9

5 Indices and tables 141

Index 143

gitter join chat

Note: This project is *work-in-progress* and is provided as-is without warranty of any kind. There may be breaking changes committed to this repository without notice.

Janis is a framework creating specialised, simple workflow definitions that are then transpiled to Common Workflow Language or Workflow Definition Language.

It was developed as part of the Portable Pipelines Project, a collaboration between:

- Melbourne Bioinformatics (University of Melbourne)
- Peter MacCallum Cancer Centre
- Walter and Eliza Hall Institute of Medical Research (WEHI).

CHAPTER 1

Introduction

Janis is a framework creating specialised, simple workflow definitions that are then transpiled to Common Workflow Language or Workflow Definition Language.

Janis requires a Python installation > 3.6, and can be installed through PIP project page by running:

```
pip3 install janis-pipelines
```

There are two ways to use Janis:

- Build workflows (and translate to CWL (v1.2) / WDL (version development) / Nextflow (in-progress))
- Run tools or workflows with CWLTool, Cromwell, or Nextflow (in progress)

1.1 Example Workflow

```
# write the workflow to `helloworld.py`
cat <<EOT >> helloworld.py
import janis as j
from janis_unix.tools import Echo

w = j.WorkflowBuilder("hello_world")

w.input("input_to_print", j.String)
w.step("echo", Echo(inp=w.input_to_print))
w.output("echo_out", source=w.echo.out)
EOT


# Translate workflow to WDL
janis translate helloworld.py wdl

# Run the workflow
janis run -o helloworld-tutorial helloworld.py --input_to_print "Hello, World!"
```

(continues on next page)

(continued from previous page)

```
# See your output
cat helloworld-tutorial/echo_out
# Hello, World!
```

1.2 How to use Janis

- Tutorial 0 - Introduction to Janis
- Tutorial 1 - Building a workflow
- Tutorial 2 - Wrapping a new tool

1.3 Workshops

In addition, there are fully self-guided workshops that more broadly go through the functionality of Janis:

- Workshop 1
- Workshop 2

1.4 Examples

Sometimes it's easier to learn by examples, here are a few hand picked examples:

- Samtools View (Docs)
- WGS Germline pipeline (GATK Only) (Docs)

CHAPTER 2

Toolbox

There are two toolboxes currently available on Janis:

- Unix <https://github.com/PMCC-BioinformaticsCore/janis-unix>‘__(‘list of tools)
- Bioinformatics <https://github.com/PMCC-BioinformaticsCore/janis-bioinformatics>‘__(‘list of tools)

2.1 References

Through conference or talks, this project has been referenced by the following titles:

- Walter and Eliza Hall Institute Talk (WEHI) 2019: *Portable Pipelines Project: Developing reproducible bioinformatics pipelines with standardised workflow languages*
- Bioinformatics Open Source Conference (BOSC) 2019: *Janis: an open source tool to machine generate type-safe CWL and WDL workflows*
- Victorian Cancer Bioinformatics Symposium (VCBS) 2019: *Developing portable variant calling pipelines with Janis*
- GIW / ABACBS 2019: *Janis: A Python framework for Portable Pipelines*
- Australian BioCommons, December 2019: *Portable pipelines: build once and run everywhere with Janis*

CHAPTER 3

Support

To get help with Janis, please ask a question on Gitter or raise an issue on GitHub.

If you find an issue with the tool definitions, please see the relevant issue page:

- Pipeline-bioinformatics

This project is work-in-progress and is still in developments. Although we welcome contributions, due to the immature state of this project we recommend raising issues through the Github issues page for Pipeline related issues.

Information about the project structure and more on contributing can be found within the documentation.

CHAPTER 4

Contents

4.1 About

This project was produced as part of the Portable Pipelines Project in partnership with:

- Melbourne Bioinformatics (University of Melbourne)
- Peter MacCallum Cancer Centre
- Walter and Eliza Hall Institute of Medical Research (WEHI)

4.1.1 Motivations

Given the awesome list of pipeline frameworks, languages and engines, why create another framework to generate workflow languages?

That's a great question, and it's a little complicated. Our project goals are to have a portable workflow specification, that is reproducible across many different compute platforms. And instead of backing one technology, we thought it would be more powerful to create a technology that can utilise the community's work.

Some additional benefits we get by writing a generic framework is we sanity check connections and also add types that exist within certain domains. For example within the bioinformatics tools, there's a *BamBai* type that represents an indexed *.bam* (+ *.bai*) file. With this framework, we don't need to worry about pesky secondary files, or the complications that come when passing them around in WDL either, this framework can take care of that.

4.1.2 Assistant

As part of the Portable Pipelines Project, we produced an execution assistant called `janis-assistant`. Its purpose is to run workflows written in Janis, track the progress and report the results back in a robust way.

4.1.3 Related project links:

	build	docs	pypi	codecov
Janis				Docs only
Janis core		See Janis		
Janis bioinformatics		See Janis		
Janis unix		See Janis		
Janis assistant		See Janis		

4.2 Pipeline concepts

This page is under construction

4.2.1 Introduction

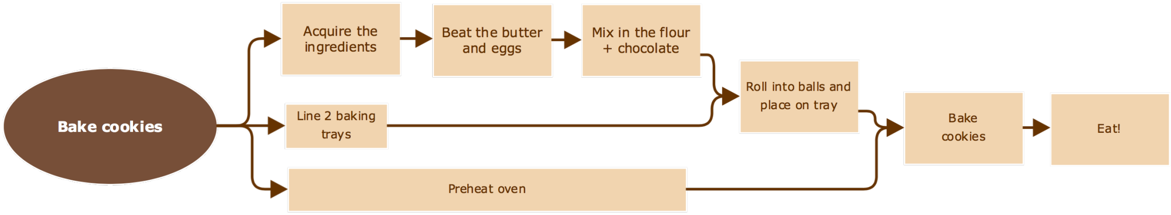
In this guide, we'll gently introduce *Workflows* and the terminology around it. More specifically, we'll discuss Pipelines in the realm of computational analysis.

4.2.2 Workflows

Before writing pipelines, it's useful to have some background knowledge of what a Workflow is. Simply put:

> *A workflow is a series of steps that are joined to each other.*

A diagram is great way to get an intuitive understanding of what a workflow is, for example to bake Chocolate chip cookies, we can refer to the following workflow:



There are 8 steps to baking (and eating) a chocolate chip cookie, each step represented by a box. The arrows represent the dependencies between the steps (what must be completed before the step can be started). For example, to beat the butter and the eggs, we must have acquired the ingredients. Or to place the rolled cookie balls onto the tray,

> we must have lined the 2 baking trays and mixed in the flour and chocolate.

We easily see how *dependencies* are represented in a workflow, and given enough resources, some of these tasks could be completed at the same time. For example, we can preheat the oven while we're preparing the cookie dough.

We'll come back to this example a few times over the course of this introduction.

4.2.3 Computational Pipelines

Computational analysis involves performing a number of processing tasks to transform some input data into a processed output. By stringing a number of these processing tasks together in a specific order, you get a pipeline.

Now, we can write simple shell script to perform a number of tasks, and this might be the way to go for simple on-off workflows. However, by spending some time to construct a pipeline, we can take advantage of some powerful computing concepts.

1. Simultaneous Job processing

   In our cookie example, we expect some of the tasks to be asynchronous (they can be performed at the same time). This is especially useful for long running tasks such as preheating the oven. If we were to script a simple (synchronous) shell script, we would have to wait for the oven to preheat to continue our baking, however a workflow engine can recognise these dependencies and run a number of these jobs at once.

2. Portability

   Additionally, shell scripts tend to be written in an explicit and non-portable way. For example you may exactly reference the location of the program you want to run which means you'd likely need to rewrite the script.

3. Fault tolerance and rerunning

   If one of the programs does not return correctly, your shell script may not correctly recognise this and continue, potentially costing your hours or days of computational time. And then when you re-run, you'll have to rerun the entire script, or modify your script to run only the failed components.

   In a workflow engine, it will automatically detect non-zero exit codes (errors), and will terminate execution of the program. Additionally, some engines are clever enough to recognise that you've entered inputs that's it's seen before, and can automatically start from the fail point.

By using Janis, we bring another set of powerful analysis tools to the table, including tests for each tool, support for transpiling to multiple commonly supported languages and type checking on all connections between tools.

4.2.4 Workflow terminology

Let's start to break down some of the terminology we've used to represent workflows.

4.2.5 Inputs and Outputs

We'll split the inputs / outputs in two ways:

1. **Step based:**

       - The inputs to a *step* is the data that specific tool needs to process.
       - The outputs to a *step* is what the *tool* produces.

2. Workflow based:

   - The inputs to a *workflow* is what we (the user) must provide to run the tools.
   - The outputs to a *workflow* is the result of processing from multiple *tools*.

To relate this back to baking cookies:

- The step 'cook' requires prepared cookie dough as an input and will produce baked cookies.
- The workflow 'Bake chocolate cookies' accepts a number of ingredients such as butter and flour as inputs.

Steps

A step is an *action* that must be performed on a set of inputs, to get the desired output.

Nesting

A step can be one tool, or we can treat a set of actions as one tool. You will recognise that a set of actions is in fact a workflow, so hence we can treat a workflow as one step (through the concept of nested workflows).

This nesting helps us to contain a set of logically related steps.

For example, when reading a book you must occasionally *turn the page*. We treat this as one action, however you can break the action of turning the page of a book into many smaller actions such as gripping the page, moving your hand and letting go of the page. However it would be cumbersome to think about performing each of these separate actions for every page in the book!

Reproducibility

Portability

4.3 Tutorial 0 - Introduction to Janis

Janis is workflow framework that uses Python to construct a declarative workflow. It has a simple workflow API within Python that you use to build your workflow. Janis converts your pipeline to the Common Workflow Language (CWL) and Workflow Description Language (WDL) for execution, and it's also great for publishing and archiving.

Janis was designed with a few points in mind:

- Workflows should be easy to build,
- Workflows and tools must be easily shared (portable),
- Workflows should be able to execute on HPCs and cloud environments.
- Workflows should be reproducible and re-runnable.

Janis uses an *abstracted execution environment*, which removes the shared file system in favour of you specifiying all the files you need up front and passing them around as a File object. This allows the same workflow to be executable on your local machine, HPCs and cloud, and we let the `execution engine` handle moving our files. This also means that we can use file systems like `S3`, `GCS`, `FTP` and more without any changes to our workflow.

> Instructions for setting up Janis on a compute cluster are under construction.

4.3.1 Requirements

- Local environment
- Python 3.6+
- Docker
- Python virtualenv (`pip3 install virtualenv`)

> **NB**: This tutorial requires Docker to be installed and available.

4.3.2 Installing Janis

We'll install Janis in a virtual environment as it preserves versioning of Janis in a reproducible way.

1. Create and activate the virtualenv:

```
# create virtual env
virtualenv -p python3 ~/janis/env
# source the virtual env
source ~/janis/env/bin/activate
```

2. Install Janis through PIP:

```
pip install -q janis-pipelines
```

3. Test that janis was installed:

```
janis -v
# --------------------  ------
# janis-core            v0.9.7
# janis-assistant       v0.9.9
# janis-unix            v0.9.0
# janis-bioinformatics  v0.9.5
# janis-pipelines       v0.9.2
# janis-templates       v0.9.4
# --------------------  ------
```

Installing CWLTool

CWLTool is a reference workflow engine for the Common Workflow Language. Janis can run your workflow using CWLTool and collect the results. For more information about which engines Janis supports, visit the Engine Support page.

```
pip install cwltool
```

Test that CWLTool has installed correctly with:

```
cwltool --version
# .../bin/cwltool 1.0.20190906054215
```

4.3.3 Running an example workflow with Janis

First off, let's create a directory to store our janis workflows. This could be anywhere you want, but for now we'll put it at `$HOME/janis/`

```
mkdir ~/janis
cd ~/janis
```

You can test run an example workflow with Janis and CWLTool with the following command:

```
janis run --engine cwltool -o tutorial0 hello
```

You'll see the `INFO` statements from CWLTool in terminal.

To see all logs, add `-d` to become:

```
janis -d run --engine cwltool -o tutorial0 hello
```

At the start, we see the two lines in our output:

```
2020-03-16T18:49:08 [INFO]: Starting task with id = 'd909df'
d909df
```

This is our workflow ID (wid) and is one way we can refer to our workflow.

After the workflow has completed (or in a different window), you can see the progress of this workflow with:

```
janis watch d909df

# WID:        d909df
# EngId:      d909df
# Name:       hello
# Engine:     cwltool
#
# Task Dir:   $HOME/janis/tutorial0
# Exec Dir:   None
#
# Status:     Completed
# Duration:   4s
# Start:      2020-03-16T07:49:08.367981+00:00
# Finish:     2020-03-16T07:49:11.881006+00:00
# Updated:    3h:51m:54s ago (2020-03-16T07:49:11+00:00)
#
# Jobs:
#     [✓] hello (1s)
#
# Outputs:
#     - out: $HOME/janis/tutorial0/out
```

There is a single output `out` from the workflow, cat-ing this result we get:

```
cat $HOME/janis/tutorial0/out
# Hello, World
```

Overriding an input

The workflow `hello` has one input `inp`. We can override this input by passing `--inp $value` onto the end of our run statement. Note the structure for workflow parameters and parameter overriding:

```
janis run <run options> worklowname <workflow inputs>
```

We can run the following command:

```
janis run --engine cwltool -o tutorial0-override hello --inp "Hello, $(whoami)"

# out: Hello, mfranklin
```

Running Janis in the background

You may want to run Janis in the background as it's own process. You could do this with `nohup [command] &`, however we can also run Janis with the `--background` flag and capture the workflow ID to watch, eg:

```
wid=$(janis run \
    --background --engine cwltool -o tutorial0-background \
    hello \
    --inp "Run in background")
janis watch $wid
```

4.3.4 Summary

- Setup a virtualenv
- Installed Janis and CWLTool
- Ran a small workflow with custom inputs

Next steps

- Workflow construction tutorial

4.4 Tutorial 1 - Building a Workflow

In this stage, we're going to build a simple workflow to align short reads of DNA.

1. Start with a pair of compressed `FASTQ` files,
2. Align these reads using `BWA MEM` into an uncompressed `SAM` file (the *de facto* standard for short read alignments),
3. Compress this into the binary equivalent `BAM` file using `samtools`, and finally
4. Sort the reads using `GATK4 SortSam`.

These tools already exist within the Janis Tool Registry, you can see their documentation online:

- BWA MEM
- Samtols View
- GATK4 SortSam

4.4.1 Preparation

To prepare for this tutorial, we're going to create a folder and download some data:

```
mkdir janis-tutorials && cd janis-tutorials

# If WGET is installed
wget -q -O- "https://github.com/PMCC-BioinformaticsCore/janis-workshops/raw/master/
↪janis-data.tar" | tar -xz

# If CURL is installed
curl -Ls "https://github.com/PMCC-BioinformaticsCore/janis-workshops/raw/master/janis-
↪data.tar" | tar -xz
```

4.4.2 Creating our file

A Janis workflow is a Python script, so we can start by creating a file called `alignment.py` and importing Janis.

```
mkdir tools
vim tools/alignment.py # or vim, emacs, sublime, vscode
```

From the `janis_core` library, we're going to import `WorkflowBuilder` and a `String`:

```
from janis_core import WorkflowBuilder, String
```

4.4.3 Imports

We have four inputs we want to expose on this workflow:

1. Sequencing Reads (`FastqGzPair` - paired end sequence)
2. Sample name (`String`)
3. Read group header (`String`)
4. Reference files (`Fasta` + index files (`FastaWithIndex`))

We've already imported the `String` type, and we can import `FastqGzPair` and `FastaWithIndex` from the `janis_bioinformatics` registry:

```
from janis_bioinformatics.data_types import FastqGzPairedEnd, FastaWithDict
```

Tools

We've discussed the tools we're going to use. The documentation for each tool has a row in the tbale caled "Python" that gives you the import statement. This is how we'll import how tools:

```
from janis_bioinformatics.tools.bwa import BwaMemLatest
from janis_bioinformatics.tools.samtools import SamToolsView_1_9
from janis_bioinformatics.tools.gatk4 import Gatk4SortSam_4_1_2
```

4.4.4 Declaring our workflow

We'll create an instance of the `WorkflowBuilder` class, this just requires a name for your workflow (can contain alphanumeric characters and underscores).

```
w = WorkflowBuilder("alignmentWorkflow")
```

A workflow has 3 methods for building workflows:

- `workflow.input` - Used for creating inputs,
- `workflow.step` - Creates a step on a workflow,
- `workflow.output` - Exposes an output on a workflow.

We give each input / step / output a unique identifier, which then becomes a node in our workflow graph. We can refer to the created node using *dot-notation* (eg: `w.input_name`). We'll see how this works in the later sections.

More information about each step will be linked from this page about the `Workflow` and `WorkflowBuilder` class.

Creating inputs on a workflow

> Further reading: Creating an input

To create an input on a workflow, you can use the `Workflow.input` method, which has the following structure:

```
Workflow.input(
    identifier: str,
    datatype: DataType,
    default: any = None,
    doc: str = None
)
```

An input requires a unique identifier (string) and a DataType (String, FastqGzPair, etc). Let's prepare the inputs for our workflow:

```
w.input("sample_name", String)
w.input("read_group", String)
w.input("fastq", FastqGzPairedEnd)
w.input("reference", FastaWithDict)
```

Declaring our steps and connections

> Further reading: Creating a step

Similar to exposing inputs, we create steps with the `Workflow.step` method. It has the following structure:

```
Workflow.step(
    identifier: str,
    tool: janis_core.tool.tool.Tool,
    scatter: Union[str, List[str], ScatterDescription] = None,
)
```

We provide a identifier for the step (unique amongst the other nodes in the workflow), and intialise our tool, passing our inputs of the step as parameters.

We can refer to an input (or previous result) using the dot notation. For example, to refer to the `fastq` input, we can use `w.fastq`.

BWA MEM

We use bwa mem's documentation to determine that we need to provide the following inputs:

- `reads`: `FastqGzPair` (connect to `w.fastq`)
- `readGroupHeaderLine`: `String` (connect to `w.read_group`)
- `reference`: `FastaWithDict` (connect to `w.reference`)

We can connect them to the relevant inputs to get the following step definition:

```
w.step(
    "bwamem",    # identifier
    BwaMemLatest(
        reads=w.fastq,
        readGroupHeaderLine=w.read_group,
        reference=w.reference
```

(continues on next page)

(continued from previous page)

```
    )
)
```

Samtools view

We'll use a very similar pattern for Samtools View, except this time we'll reference the output of `bwamem`. From bwa mem's documentation, there is one output called `out` with type `Sam`. We'll connect this to `SamtoolsView` only input, called `sam`.

```
w.step(
    "samtoolsview",
    SamToolsView_1_9(
        sam=w.bwamem.out
    )
)
```

SortSam

In addition to connecting the output of `samtoolsview` to Gatk4 SortSam, we want to tell SortSam to use the following values:

- sortOrder: `"coordinate"`
- createIndex: `True`

Instead of connecting an input or a step, we just just provide the literal value.

```
w.step(
    "sortsam",
    Gatk4SortSam_4_1_2(
        bam=w.samtoolsview.out,
        sortOrder="coordinate",
        createIndex=True
    )
)
```

Exposing outputs

> Further reading: Creating an output

Outputs have a very similar syntax to both inputs and steps, they take an `identifier` and a named `source` parameter. Here is the structure:

```
Workflow.output(
    identifier: str,
    datatype: DataType = None,
    source: Node = None,
    output_folder: List[Union[String, Node]] = None,
    output_name: Union[String, Node] = None
)
```

Often, we don't want to specify the output data type, because we can let Janis do this for us. We'll talk about the `output_folder` and `output_name` in the next few sections. For now, we just have to specify an output identifier and a source.

```
w.output("out", source=w.sortsam.out)
```

4.4.5 Workflow + Translation

Hopefully you have a workflow that looks like the following!

```
from janis_core import WorkflowBuilder, String

from janis_bioinformatics.data_types import FastqGzPairedEnd, FastaWithDict

from janis_bioinformatics.tools.bwa import BwaMemLatest
from janis_bioinformatics.tools.samtools import SamToolsView_1_9
from janis_bioinformatics.tools.gatk4 import Gatk4SortSam_4_1_2

w = WorkflowBuilder("alignmentWorkflow")

# Inputs
w.input("sample_name", String)
w.input("read_group", String)
w.input("fastq", FastqGzPairedEnd)
w.input("reference", FastaWithDict)

# Steps
w.step(
    "bwamem",
    BwaMemLatest(
        reads=w.fastq,
        readGroupHeaderLine=w.read_group,
        reference=w.reference
    )
)
w.step(
    "samtoolsview",
    SamToolsView_1_9(
        sam=w.bwamem.out
    )
)

w.step(
    "sortsam",
    Gatk4SortSam_4_1_2(
        bam=w.samtoolsview.out,
        sortOrder="coordinate",
        createIndex=True
    )
)

# Outputs
w.output("out", source=w.sortsam.out)
```

We can translate the following file into Workflow Description Language using janis from the terminal:

```
janis translate tools/alignment.py wdl
```

4.4.6 Running the alignment workflow

We'll run the workflow against the current directory.

```
janis run -o . --engine cwltool \
    tools/alignment.py \
    --fastq data/BRCA1_R*.fastq.gz \
    --reference reference/hg38-brca1.fasta \
    --sample_name NA12878 \
    --read_group "@RG\tID:NA12878\tSM:NA12878\tLB:NA12878\tPL:ILLUMINA"
```

After the workflow has run, you'll see the outputs in the current directory:

```
ls

# drwxr-xr-x  mfranklin  1677682026   160B  data
# drwxr-xr-x  mfranklin  1677682026   256B  janis
# -rw-r--r--  mfranklin  wheel        2.7M  out.bam
# -rw-r--r--  mfranklin  wheel        296B  out.bam.bai
# drwxr-xr-x  mfranklin  1677682026   320B  reference
# drwxr-xr-x  mfranklin  1677682026   128B  tools
```

OPTIONAL: Run with Cromwell

If you have `java` installed, Janis can run the workflow in the Crowmell execution engine by using the `--engine cromwell` parameter:

```
janis run -o run-with-cromwell --engine cromwell \
    tools/alignment.py \
    --fastq data/BRCA1_R*.fastq.gz \
    --reference reference/hg38-brca1.fasta \
    --sample_name NA12878 \
    --read_group "@RG\tID:NA12878\tSM:NA12878\tLB:NA12878\tPL:ILLUMINA"
```

4.5 Tutorial 2 - Wrapping a new tool

> This tutorial builds on the content and output from Tutorial 1.

4.5.1 Introduction

A CommandTool is the interface between Janis and a program to be executed. Simply put, a CommandTool has a name, a command, inputs, outputs and a container to run in. Inputs and arguments can have a prefix and / or position, and this is used to construct the command line.

The Janis documentation for the CommandTool gives an introduction to the tool structure and a template for constructing your own tool. A tool wrapper must provide all of the information to configure and run the specified tool, this includes the `base_command`, janis.ToolInput, janis.ToolOutput, a `container` and its version.

Container

> *Further information*: Containerising your tools

For portability, Janis requires that you specify an OCI compliant `container` (eg: Docker) for your tool. Often there will already be a container with some searching, however here's a guide on preparing your tools in containers to ensure it works across all environments.

4.5.2 Preparation

The sample data to test this tool is computed in Tutorial 1. You can follow this tutorial, but running the example will require you to have completed and obtained the bam from the first tutorial.

4.5.3 Samtools flagstat

In this tutorial we're going to wrap the `samtools flagstat` tool - flagstat counts the number of alignments for each FLAG type within a bam file.

Samtools project links

- Latest version: `1.9`
- Project page: http://www.htslib.org/doc/samtools.html
- Github: samtools/samtools
- Docker containers: quay.io/biocontainers/samtools (automatically / community generated)
  - Latest tag: `1.9--h8571acd_11`

Command to build

We want to replicate the following command for `Samtools Flagstat` in Janis:

```
samtools flagstat [--threads n] <in.bam>
```

Hence, we can isolate the following information:

- Base commands: `"samtools"`, `"flagstat"`
- The positional `<in.bam>` input
- The configuration `--threads` input

Command tool template

The following template is the minimum amount of information required to wrap a tool. For more information, see the CommandToolBuilder documentation.

> We've removed the optional fields: `tool_module`, `tool_provider`, `metadata`, `cpu`, `memory` from the following template.

We're going to use `Bam` and `TextFile` data types, so let's import them as well.

```
from janis_core import CommandToolBuilder, ToolInput, ToolOutput, Int, Stdout

ToolName = CommandToolBuilder(
    tool: str="toolname",
    base_command=["base", "command"],
    inputs=[],  # List[ToolInput]
    outputs=[], # List[ToolOutput]
    container="container/name:version",
    version="version"
)
```

Tool information

Let's start by creating a file with this template inside a second output directory:

```
mkdir -p tools
vim tools/samtoolsflagstat.py
```

We can start by filling in the basic information:

- Rename the variable (ToolName) to be `SamtoolsFlagstat`
- Fill the parameters:
  - `tool`: A unqiue tool identifier, eg: `"SamtoolsFlagStat"`.
  - `base_command` to be `["samtools", "flagstat"]`
  - `container` to be `"quay.io/biocontainers/samtools:1.9--h8571acd_11"`
  - `version` to be `"v1.9.0"`

You'll have a class definition like the following

```
SamtoolsFlagstat = CommandToolBuilder(
    tool: str="samtoolsflagstat",
    base_command=["samtools", "flagstat"],
    container="quay.io/biocontainers/samtools:1.9--h8571acd_11",
    version="1.9.0",

    inputs=[],  # List[ToolInput]
    outputs=[] # List[ToolOutput]
)
```

Inputs

> Further reading: `ToolInput`

We'll use the ToolInput class to represent these inputs. A `ToolInput` provides a mechanism for binding this input onto the command line (eg: prefix, position, transformations). See the documentation for more ways to configure a ToolInput.

```
janis.ToolInput(
        tag: str,
        input_type: DataType,
        position: Optional[int] = None,
        prefix: Optional[str] = None,
```

(continues on next page)

(continued from previous page)

```
    # more configuration options
        separate_value_from_prefix: bool = None,
        prefix_applies_to_all_elements: bool = None,
        presents_as: str = None,
        secondaries_present_as: Dict[str, str] = None,
        separator: str = None,
        shell_quote: bool = None,
        localise_file: bool = None,
        default: Any = None,
        doc: Optional[str] = None
)
```

> Nb: A ToolInput must have a `position` OR `prefix` in order to be bound onto the command line. If the prefix is specified with no position, a `position=0` is automatically applied.

Now we can declare our two inputs:

1. Positional bam input
2. Threads configuration input with the prefix `--threads`

We're going to give our inputs a name through which we can reference them by. This allows us to specify a value from the command line, or connect the result of a previous step within a workflow.

```
SamtoolsFlagstat = CommandToolBuilder(
    # ... tool information
    inputs=[
        # 1. Positional bam input
        ToolInput(
            "bam",      # name of our input
            Bam,
            position=1,
            doc="Input bam to generate statistics for"
        ),
        # 2. `threads` inputs
        ToolInput(
            "threads",  # name of our input
            Int(optional=True),
            prefix="--threads",
            doc="(-@)  Number of additional threads to use [0]"
        )
    ],
    # outputs
```

Outputs

> Further reading: `ToolOutput`

We'll use the ToolOutput class to collect and represent these outputs. A `ToolOutput` has a type, and if not using `stdout` we can provide a `glob` parameter.

```
janis.ToolOutput(
    tag: str,
    output_type: DataType,
    glob: Union[janis_core.types.selectors.Selector, str, None] = None,
    # more configuration options
    presents_as: str = None,
```

(continues on next page)

(continued from previous page)

```
    secondaries_present_as: Dict[str, str] = None,
    doc: Optional[str] = None
)
```

The only output of `samtools flagstat` is the statistics that are written to `stdout`. We give this the name `"stats"`, and collect this with the `Stdout` data type. We can additionally tell Janis that the Stdout has type `TextFile`.

```
SamtoolsFlagstat = CommandToolBuilder(
    # ... tool information + inputs
    outputs=[
        ToolOutput("stats", Stdout(TextFile))
    ]
)
```

Tool definition

Putting this all together, you should have the following tool definition:

```
from janis_core import CommandToolBuilder, ToolInput, ToolOutput, Int, Stdout

from janis_unix.data_types import TextFile
from janis_bioinformatics.data_types import Bam

SamtoolsFlagstat = CommandToolBuilder(
    tool="samtoolsflagstat",
    base_command=["samtools", "flagstat"],
    container="quay.io/biocontainers/samtools:1.9--h8571acd_11",
    version="v1.9.0",
    inputs=[
        # 1. Positional bam input
        ToolInput("bam", Bam, position=1),
        # 2. `threads` inputs
        ToolInput("threads", Int(optional=True), prefix="--threads"),
    ],
    outputs=[ToolOutput("stats", Stdout(TextFile))],
)
```

4.5.4 Testing the tool

We can test the translation of this from the CLI:

> If you have multiple command tools or workflows declared in the same file, you will need to provide the `--name` parameter with the name of your tool.

```
janis translate tools/samtoolsflagstat.py wdl # or cwl
```

In the following translation, we can see the WDL representation of our tool. In particular, the `command` block gives us an indication of how the command line might look:

```
task samtoolsflagstat {
  input {
    Int? runtime_cpu
```

(continues on next page)

(continued from previous page)

```
    Int? runtime_memory
    File bam
    Int? threads
  }
  command <<<
    samtools flagstat \
      ~{"--threads " + threads} \
      ~{bam}
  >>>
  runtime {
    docker: "quay.io/biocontainers/samtools:1.9--h8571acd_11"
    cpu: if defined(runtime_cpu) then runtime_cpu else 1
    memory: if defined(runtime_memory) then "~{runtime_memory}G" else "4G"
    preemptible: 2
  }
  output {
    File stats = stdout()
  }
}
```

Running the workflow

A reminder that the sample data for this section requires you to have completed Tutorial 1.

We can call the `janis run` functionality, and use the output from tutorial1:

```
janis run -o tutorial2 tools/samtoolsflagstat.py --bam tutorial1/out.bam
```

OUTPUT:

```
WID:        f9e89f
EngId:      f9e89f
Name:       samtoolsflagstatWf
Engine:     cwltool

Task Dir:   $HOME/janis-tutorials/tutorial2/
Exec Dir:   $HOME/janis-tutorials/tutorial2/janis/execution/

Status:     Completed
Duration:   4s
Start:      2019-11-14T04:51:59.744526+00:00
Finish:     2019-11-14T04:52:03.869735+00:00
Updated:    Just now (2019-11-14T04:52:05+00:00)

Jobs:
    [✓] samtoolsflagstat (N/A)

Outputs:
    - stats: $HOME/janis-tutorials/tutorial2/stats.txt
```

Janis (and CWLTool) said the tool executed correctly, let's check the output file:

```
cat tutorial2/stats.txt
```

```
20061 + 0 in total (QC-passed reads + QC-failed reads)
0 + 0 secondary
337 + 0 supplementary
0 + 0 duplicates
19971 + 0 mapped (99.55% : N/A)
19724 + 0 paired in sequencing
9862 + 0 read1
9862 + 0 read2
18606 + 0 properly paired (94.33% : N/A)
19544 + 0 with itself and mate mapped
90 + 0 singletons (0.46% : N/A)
860 + 0 with mate mapped to a different chr
691 + 0 with mate mapped to a different chr (mapQ>=5)
```

4.6 Tutorial 3 - Naming and organising outputs

Sometimes it's useful to organise your outputs in specific ways, especially when Janis might need to interact with other specific systems. By default, outputs are named by the tag of the output. The extension is derived from the type of the output (the Bam filetype knows it uses a `.bam` extension).

For example, if you had an output called `out` which had type `BamBai`, your output files by default would have the name `out.bam` and `out.bam.bai`.

When exposing an output on a workflow, there are two arguments you can provide to override this behaviour:

- `output_name: Union[str, InputSelector, InputNode] = None`
- `output_folder: Union[str, Tuple[Node, str], List[Union[str, InputSelector, Tuple[Node, str]]]] = None`

4.6.1 Preparation

This tutorial uses the workflow build in Tutorial 1. You can follow this guide, but the example will require you to have built (or acquired) the workflow from tutorial 1.

4.6.2 Output name

Simply put, `output_name` is the dervied filename of the output without the extension. By default, this is the `tag` of the output.

You can specify a new output name in 2 ways:

1. A static string: `output_name="new name for output"`
2. Selecting an input value (given "sample_name" is the name of an input):
   1. `output_name=workflow.sample_name`
   2. `output_name=InputSelector("sample_name")`

You should make the following considerations:

- The input you select should be a string, or
- If the output you're naming is an array, the input you select should either be:
  - singular

  - have the same number of elements in it.

Janis will either fall back to the first element if it's a list, or default to the output tag. This may cause outputs to override each other.

4.6.3 Output folder

Similar to the output name, the `output_folder` is folder, or group of nested folders into which your output will be written. By default, this field has no value and outputs are linked directly into the output directory.

If the output_folder field is an array, a nested folder is created for each element in ascending order (eg: `["parent", "child", "child_of_child"]`).

There are multiple ways to specify output directories:

1. A static string: `output_folder="my_outputs"`
2. Selecting an input value (given "sample_name" is the name of an input):
   1. `output_folder=workflow.sample_name`
   2. `output_folder=InputSelector("sample_name")`
3. An array of a combination of values:
   - `output_folder=["variants", "unmerged"]`
   - `output_folder=["variants", w.sample_name]`
   - `output_folder=[w.other_field, w.sample_name]`

4.6.4 Alignment workflow

In our alignment example (`tools/alignment.py`), we have the following outputs:

```
w.output("out", source=w.sortsam.out)
w.output("stats", source=w.flagstat.stats)
```

We want to name the output in the following way:

- Grouped the bam `out` into the folder: `bams`,
- Group the samtools summary into folder `statistics`
- Both outputs named: `sample_name` (Janis will automatically add the `.bam` or `.txt` extension).

```
w.output(
    "out",
    source=w.sortsam.out,
    output_name=w.sample_name,
    output_folder=["bams"]
)
w.output(
    "stats",
    source=w.flagstat.stats,
    output_name=w.sample_name,
    output_folder=["statistics"]
)
```

Run the workflow again and inspect the new results:

```
janis translate tools/alignment.py wdl
janis run -o tutorial3 tools/alignment.py \
    --fastq data/BRCA1_R*.fastq.gz \
    --reference reference/hg38-brca1.fasta \
    --sample_name NA12878 \
    --read_group "@RG\tID:NA12878\tSM:NA12878\tLB:NA12878\tPL:ILLUMINA"
```

And then after the workflow has finished, we find the following output structure:

```
$ ls tutorial3/*

tutorial3/bams:
-rw-r--r--  PMCI\Bioinf-Cluster   2.7M NA12878.bam
-rw-r--r--  PMCI\Bioinf-Cluster   296B NA12878.bam.bai

tutorial3/statistics:
-rw-r--r--  PMCI\Bioinf-Cluster   408B NA12878.txt

tutorial3/janis:
[...ommitted]
```

4.7 Adding a tool to a toolbox

> This section is optional.

This section requires that you have Git configured, and a GitHub account.

4.7.1 Fork

> A fork is a copy of a repository. Forking a repository allows you to freely experiment with changes without affecting the original project. - GitHub: Fork a Repo
>
> Further reading: Fork an example repository

If you want to add tools to an existing toolbox, you could for example,

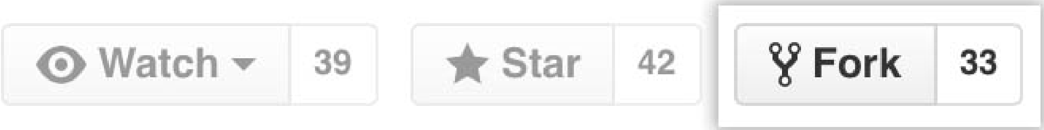
- Fork janis-unix.
- Add and test new tools,
- Create a Pull Request to integrate your changes.

To create a fork of `janis-unix`:

1. Navigate to the Janis-Unix repository:

```
https://github.com/PMCC-BioinformaticsCore/janis-unix
```

2. In the top-right corner of the page, click **Fork**.



Fork a repository

Clone your fork

```
# 1. Clone your repiostory
git clone https://github.com/yourusername/janis-unix.git

# 2. Go into the checked out code
cd janis-bioinformatics

# 3. Add the original repo as 'upstream'
git remote add upstream https://github.com/PMCC-BioinformaticsCore/janis-unix.git
```

Keeping your fork up to date

> Further reading: Syncing a fork

```
# 1. Get the changes from the original changes
git fetch upstream master

# 2. Merge these changes into your current branch
git merge upstream/master

# 3. Push these updates to your fork
git push
```

You may have to deal with conflicts if changes in the upstream have modified the same files you've locally modified.

4.7.2 Making a change

First off, make sure your fork is up to date. Then let's checkout to a new branch:

```
git checkout -b add-greet-tool
```

Our new tool is going to simply print `"Hello, " + $name` to the console.

```
vim janis_unix/tools/greet.py
```

```
from janis_core import String, ToolInput, ToolOutput, ToolArgument, Stdout, Boolean
from .unixtool import UnixTool


class Greet(UnixTool):
    def tool(self):
        return "greet"

    def friendly_name(self):
        return "Greet"

    def base_command(self):
        return "echo"

    def arguments(self):
        return [ToolArgument("Hello, ", position=0)]

    def inputs(self):
```

(continues on next page)

(continued from previous page)

```
        return [ToolInput("name", String(), position=1)]

    def outputs(self):
        return [ToolOutput("out", Stdout())]
```

Let's make sure we add the Greet tool to be exported by this toolbox:

```
vim janis_unix/tools/__init__.py
```

```
# ...other imports
from .hello import HelloWorkflow
from .greet import Greet
```

Install the repository

The Janis environment at Peter Mac has been configured to allow your locally installed modules to override the modules from within the python environment.

You just have to reference the version of `pip3` outside of the Janis Python env: `/usr/bin/pip3`.

> Don't forget the `.` at the end of the pip install!

```
# --no-dependencies flag ensures we don't install janis core / assistant
/usr/bin/pip3 install --no-dependencies --user .
```

Testing the change

Let's test the tool we just added:

```
janis spider Greet
```

Voila! We added a tool to the janis-unix toolbox.

4.7.3 Commit and push your changes

Let's commit our changes!

```
# 0. Check the status of your changes
git status

# 1. Stage our changes to commit (all of them)
git add .

# 2. Commit with a message
git commit -m "Adds new tool 'Greet'"
```

We want to push your branch `add-greet-tool` to the remote to create a pull request.

```
# Set the upstream on our branch, and push to remote 'origin'
git push --set-upstream origin add-greet-tool

Total 0 (delta 0), reused 0 (delta 0)
```

(continues on next page)

(continued from previous page)

```
remote:
remote: Create a pull request for 'add-greet-tool' on GitHub by visiting:
remote:      https://github.com/yourname/janis-unix/pull/new/add-greet-tool
remote:
To github.com:yourname/janis-unix.git
 * [new branch]      add-greet-tool -> add-greet-tool
Branch 'add-greet-tool' set up to track remote branch 'add-greet-tool' from 'origin'.
```

Create a pull request by clicking the link:

- https://github.com/yourname/janis-unix/pull/new/add-greet-tool

4.8 Running your workflow

Now you've built your workflow, let's run it on your computer.

_We're working on a runner for engine that allows you to run your favourite workflow engine without going through this manual export process. Stay tuned for more!

4.8.1 Overview

In this tutorial, we're going to run the workflow you have built with `janis-runner`. Behind the scenes this uses Cromwell or CWLTool to manage the execution.

Engines

To run the workflow, we're going to use a workflow execution engine, or just *engine* for short. Some engines only accept certain workflow specifications, for example the CWL reference engine (cwltool) only accepts CWL, while Cromwell (developed by the Broad Institute) accepts both WDL and CWL.

4.8.2 What you'll need

To complete this tutorial, you'll need to have an installation of janis (see the Getting Started guide for more info), a workflow to run and an engine available.

Execution Engine

CWLTool

`cwltool` is the reference engine for CWL. We'd recommend visiting the install guide, however you can install `cwltool` through Pip by running:

```
pip3 install cwltool
```

WDL

For WDL, we'd recommend `Cromwell`, developed by the Broad Institute and developed alongside the *openwdl* community. You can find their docs here. Cromwell is packaged as a `jar` file, you must have a Java 8 runtime installed (ie, running `java -version` in the console gives you `1.8.0` or higher).

Janis will automatically download Cromwell, or it will automatically detect versions in `~/.janis/`. It's possible to configure Cromwell to run in a variety of ways through the configuration guide.

Containerisation

Most commonly, people use `docker` which both `cwltool` and `Cromwell` support by default. Some HPC friendly alternatives include`singularity` and `udocker`; although these engines can be configured independently with both of these user-space replacements, we're still working on an easy way to configure Janis to use these.

4.8.3 Let's get started

We're going to use the Janis command-line interface (CLI) to run these workflows and track their progress. For this example, we're going to run the *Hello* workflow and put the relevant output files in an output directory called `hello_dir`.

By default, Janis will run the workflow using CWLTool, this can be changed by passing the `--engine` parameter with either `"cwltool"` or `"cromwell"`.

Let's do that now:

```
janis run --engine [cwltool|cromwell] -o hello_dir hello
```

We'll see some important information in our console:

- The ID of our task. This is useful to gain access to our workflow information at a later date.

```
2019-09-26T00:25:00+00:00 [INFO]: Starting task with id = 'a4f047'
```

- The metadata screen, this tells us about the progress of our workflow.
  - There are 3 status indicators:
    - `[...]`: processing
    - `[~]`: running
    - `[✓]`: completed
    - `[!]`: failed

```
TID:        a4f047
WID:        1ca1d418-df9b-45ea-b6b8-2d210be310b3
Name:       hello

Engine:     cromwell
Engine url: localhost:56232

Task Dir:   /Users/franklinmichael/janis/execution/hello/20190926_102526_a4f047/
Exec Dir:   /Users/franklinmichael/cromwell-executions/hello/1ca1d418-df9b-45ea-b6b8-
↪2d210be310b3
```

(continues on next page)

(continued from previous page)

```
Status:      Completed
Duration:    17
Start:       2019-09-26T00:25:55.771000+00:00
Finish:      2019-09-26T00:26:13.079000+00:00

Jobs:
    [✓] hello.hello (13s :: 76.5 %)

...

Finished managing task 'a4f047'. View the task outputs: file:///Users/franklinmichael/
↪janis/execution/hello/20190926_102526_a4f047/
a4f047
```

If we look at the contents of our output directory (`$task dir + "/outputs/"`, eg: `/Users/franklinmichael/janis/execution/hello/20190926_102526_a4f047/outputs/`), we see one file called `hello.out` which contains the string “Hello, World!”.

Overriding workflow inputs

The `hello` workflow allows us to override the string that gets printed to the console. We can see in the `hello` documentation, there is an input called `inp` (optional string) that we can override.

There are two ways to override inputs in Janis:

1. Creating a yaml job file: `janis inputs hello > hello-job.yml`, this file can be modified and provided to the run with `--inputs hello-job.yml`.
2. Parameter overrides within the run command.

We’re going to take the second route by providing `"Hello, Janis!"` to the workflow:

```
janis run hello --inp "Hello, Janis!"
```

Outputting the result in the outputs folder yields:

```
Hello, Janis!
```

Success!

4.8.4 Finished!

Congratulations on running your workflow! You can now get more advanced with the following tutorials:

- Building a simple bioinformatics workflow
- Building tools

4.8.5 Advanced arguments

CPU and Memory overrides

- `runtime_cpu: int` is the number of cpus that are required.

- `runtime_memory`: `float` is specified in number of `Gigabytes`.
- `runtime_disks`: `string` is specified in the format `target size type` (eg: `"local-disk 100 SSD"`)

When exporting the workflow, you have the ability to generate schema that allows you to override the `cpu` and `memory` values at a workflow inputs level. That is, you can include parameters in your input file that will ask the engine to run your workflow with specific memory and cpu requirements.

You can generate a workflow with resource overrides like so:

```
w.translate("cwl", with_resource_overrides=True, **other_kwargs)
```

You can generate an inputs file with the overrides withe the following:

```
def generate_inputs_override(
    additional_inputs=None,
    with_resource_overrides=False,
    hints=None
):
    pass
```

or with the CLI:

```
janis inputs myworkflow.py [--resources] > job.yaml
```

In CWL, this looks like:

```
taskname_runtime_cpu: null
taskname_runtime_memory: null
subworkflowname_task2name_runtime_cpu: null
subworkflowname_task2name_runtime_memory: null
```

And in WDL:

```
{
    "$name.taskname_runtime_memory": null,
    "$name.taskname_runtime_cpu": null,
    "$name.taskname_runtime_disks": null,
    "$name.subworkflowname_task2name_runtime_cpu": null,
    "$name.subworkflowname_task2name_runtime_cpu": null,
    "$name.subworkflowname_task2name_runtime_disks": null
}
```

Overriding export location

> Default: `export_path='.'`

You can override the export path to a more convenient location, by providing the ‘‘parameter to `translate`. Prefixing the path with ~ (*tilde*) will replace this per `os.path.expanduser`. You can also use the following placeholders (see `github/janis/translation/exportpath` for more information):

- `"{language}"` - ‘cwl’ or ‘wdl’
- `"{name}"` - Workflow identifier

You can output a workflow through the CLI with the following commmand:

```
janis translate myworkflow.py [cwl|wdl] --output-dir /path/to/outputdir/
```

4.9 Containerising a tool

Portability and reproducibility are important aspects of `janis`. Building and using containers for your tool is a great way to ensure compatibility and portability across institutes.

4.9.1 Getting started

In this tutorial, we're going to introduce containers and go through different ways to find or build containers for your tool.

What you'll need

You'll need an installation of Docker, and a Dockerhub account for uploading your tool to the cloud. Other cloud-container vendors such as quay.io may be used instead of Dockerhub.

You'll also need your tool, the dependencies and know how to install it.

Licenses and warnings

You should have all relevant permission to be able to distribute the tool that you are wrapping. Most open source licenses allow you distribute a compiled or derived version, but may have some restrictions.

Versioning

It's important that we consider the versions of our tool when we construct our container. A large factor of this project is reproducibility, and this is only possible if we use the same tools. We highly discourage the use of the `latest` tag, and instead tag the container with its explicit version.

> Note: The same docker tag doesn't ensure you get the same container each time as a user can upload a new build for the same tag. Ideally you could use the Docker `digest` however not all container software support this.

4.9.2 Let's get started!

What are containers

Containers are allow an operating system and software to be virtualised on a different computer (the host). Although containers can be used for a variety of purposes, we use them as a "sandbox" for us to run analysis in. The important aspect for us is that we can guarantee reproducibility with the same container across compute platforms.

In this guide we'll be creating Docker containers, however other virtualisation software (such as Singularity, uDocker or Shifter) can read this common (OCI compliant) standard.

How do I get a container?

When we look for a container, we want to make sure it's high quality and is as described. Some services (such as the community supported Biocontainers) produce high quality containers.

Here is a number of criteria we're looking for in a container:

- From a reputable source.
- The Dockerfile is available (gives us trust that there's not malicious software in there).
- The versions available are sensible and match the software versions.
  - A container with only the `latest` tag does not satisfy this criteria.

Here is our recommendation for finding a container given the listed criteria:

1. Look for a container from the tool provider (maybe through their GitHub).
2. Look for a container from a reputable source on Dockerhub or quay.io.
3. Find a third-party container that meets all the criteria above.

If you are unable to find a container, you might need to build one (and maybe find a tech savvy colleague to help you).

4.9.3 Building a container

We're going to build a Docker container as at the moment they are the most portable option. Each Docker container has a recipe, called a `Dockerfile`, it's a special file that tells Docker how a container is built, and how it should be executed.

Each container should have a base, ie an operating system or even another docker container.

Here are the following types of containers we'll create:

- *Python module*
- *Python script*
- *Downloadable binary w/ requirements*
- *Makeable program*

Basic setup

For all of the following guides, you'll need to create a folder for your Dockerfile and other dependencies. It's important you place your dependencies within this directory (or subdirectories) to avoid bloating your build context.

```
tooldir=mytool-docker
mkdir $tooldir && cd $tooldir
touch Dockerfile
```

Python module

We're going to package up a Python module that you can run from the console. For this example we'll wrap Janis in a container.

> Within the `setup.py`, it has a dictionary value within the `entry_points` kwarg and a value for the `console_script` key. For example, Janis-assitant setup.py:24 looks like this:

```
entry_points={
    "console_scripts": ["janis=janis_assistant.cli:process_args"],
}
```

Given our application is compatible with Python 3.7, we simply `pip install` our module on top, place this in our `Dockerfile`:

```
FROM python:3.7
RUN pip install janis-pipelines
CMD janis
```

> Although `janis-pipelines` is compatible with the alpine Python 3.7 container, we'll use the full version, see "The best Docker base image for your Python application" for more information.

We can build this container with the following command which will

- Use the build context: `.`
- *Automatically* build the file called `Dockerfile` relative to the build context
- Give the container the tag: `yourname/packagename`

```
docker build -t yourname/janis-pipelines .
```

We can test the container works by running:

```
docker run yourname/janis-pipelines janis -v
#--------------------  ------
#janis-core            v0.7.1
#janis-assistant       v0.7.7
#janis-unix            v0.7.0
#janis-bioinformatics  v0.7.1
#--------------------  ------
```

OR:

```
docker run -it --entrypoint /bin/sh yourname/janis-pipelines
## inside the container
$ janis translate hello wdl
# translation here
$ exit # exit the container
```

Python script

We have a single python script that we want to run inside a Docker container. This time we're going to add our files to the `/opt` directory (in the container), add that directory to the `$PATH` variable and then we can simply call your docker container.

We're going to create a little Python program to write something to the console and then end. Save the following Python program (including the Shebang) into the directory with our Dockerfile.

`hello.py`:

```
#!/usr/bin/env python3
print("Hello, World")
```

We need to make `hello.py` executable by:

```
chmod +x hello.py
```

We can test this worked by calling our script, by running `./hello.py` in our terminal

We'll now edit our `Dockerfile` with the following steps:

- Use a `python:3.7` base.
- Add the file into container.
- Modify the path to include the script directory (`/opt`).
- Use the `hello.py` script an entry-point.

```
FROM python:3.7.5-alpine
ADD hello.py /opt/
ENV PATH="/opt:${PATH}"
CMD hello.py
```

We can build this container with the following command:

Don't forget the full stop `.` at the end of the docker build command

```
docker build -t yourname/hello .
```

You can test that it worked in two ways:

1. Run the container (using the entrypoint (CMD) `hello.py`):

```
docker run yourname/hello
# Hello, World!
```

1. Go into the container, and run the script:

```
docker run -it --entrypoint /bin/sh yourname/hello
## inside the container
$ hello.py
# Hello, World!
$ exit # exit the container
```

Downloadable Binary with Requirements

This section is still under construction.

This style of container is very similar to the Python script. We'll `ADD` our binary (potentially from the web) to `/opt/toolname`, add this to the path and add an entry point.

You will need to consider which Base OS to use, for now we'll use alpine as it's very lightweight, but you might need to consider `ubuntu` or `centos`. We'll assume that your tool `mytool` runs out of a `bin` folder.

`Dockerfile`

```
FROM alpine:latest
RUN mkdir -p /opt/mytool/
ADD https://mytool.net/releases/tool.zip /opt/
ENV PATH="/opt/mytool/bin:${PATH}"        # assume ./mytool/bin/
CMD mytool
```

Makeable Program

This section is under construction, please refer to an example Samtools Docker for more information:

Dockerfile

```
FROM ubuntu:16.04

MAINTAINER Michael Franklin <michael.franklin@petermac.org>

RUN apt-get update -qq \
  && apt-get install -qq bzip2 gcc g++ make zlib1g-dev wget libncurses5-dev liblzma-
↪dev libbz2-dev

ENV SAMTOOLS_VERSION 1.9

LABEL \
  version="${SAMTOOLS_VERSION}" \
  description="Samtools image for use in Workflows"

RUN cd /opt/ \
        && wget https://github.com/samtools/samtools/releases/download/${SAMTOOLS_
↪VERSION}/samtools-${SAMTOOLS_VERSION}.tar.bz2 \
        && tar -xjf samtools-${SAMTOOLS_VERSION}.tar.bz2 \
        && rm -rf samtools-${SAMTOOLS_VERSION}.tar.bz2  \
        && cd samtools-${SAMTOOLS_VERSION}/ \
        && make && make install

ENV PATH="/opt/samtools-${SAMTOOLS_VERSION}/:${PATH}"
```

4.9.4 Additional helpful hints

Keeping Dockerfiles separate from source code

You can keep your Dockerfile in a different place to your file (with restrictions) however you must change a few arguments:

- Your build context (the last argument) should be in a place that can access both the Dockerfile and program files as subdirectories
- The `COPY hello.py` command in the Dockerfile must be changed to be relative to the build context.
- You can then supply `-f` argument to point to the Dockerfile, relative to the build context:

Example: If the python `hello.py` file is stored in a subdirectory called `src`, and the Dockerfile is stored in a subdirectory called `docker_stuff`, you could use the following:

Dockerfile:

```
COPY src/hello.py /install_dir/hello.py
```

Build:

Change into the parent directory of `src/` and `docker_stuff/`

```
docker build -t yourname/hello -f dockerstuff/Dockerfile .
```

The less steps your run, the smaller the container

By joining RUN commands together using the && operator, your total containers may be smaller in size.

4.10 Pipelines

4.11 Tools

Automatically generated index page for Tools:

4.12 Data Types

Automatically generated index page for Data Types:

4.12.1 BAI

Index of the BAM file (https://www.biostars.org/p/15847/), http://software.broadinstitute.org/software/igv/bam

Quickstart

```python
from janis_bioinformatics.data_types.bai import Bai

w = WorkflowBuilder("my_workflow")

w.input("input_bai", Bai(optional=False))

# ...other workflow steps
```

This page was automatically generated on 2020-12-22.

4.12.2 BAM

A binary version of a SAM file, http://software.broadinstitute.org/software/igv/bam

Quickstart

```python
from janis_bioinformatics.data_types.bam import Bam

w = WorkflowBuilder("my_workflow")

w.input("input_bam", Bam(optional=False))

# ...other workflow steps
```

This page was automatically generated on 2020-12-22.

4.12.3 bed

A local file

Quickstart

```
from janis_bioinformatics.data_types.bed import Bed

w = WorkflowBuilder("my_workflow")

w.input("input_bed", Bed(optional=False))

# ...other workflow steps
```

This page was automatically generated on 2020-12-22.

4.12.4 BedGz

A gzipped file

Quickstart

```
from janis_bioinformatics.data_types.bed import BedGz

w = WorkflowBuilder("my_workflow")

w.input("input_bedgz", BedGz(optional=False))

# ...other workflow steps
```

This page was automatically generated on 2020-12-22.

4.12.5 BedTABIX

A gzipped file

Secondary Files

- `.tbi`

Note: For more information, visit Secondary / Accessory Files

Quickstart

```
from janis_bioinformatics.data_types.bed import BedTabix

w = WorkflowBuilder("my_workflow")

```

(continues on next page)

(continued from previous page)

```
w.input("input_bedtabix", BedTabix(optional=False))

# ...other workflow steps
```

This page was automatically generated on 2020-12-22.

4.12.6 Boolean

A boolean

Quickstart

```
from janis_core.types.common_data_types import Boolean

w = WorkflowBuilder("my_workflow")

w.input("input_boolean", Boolean(optional=False))

# ...other workflow steps
```

This page was automatically generated on 2020-12-22.

4.12.7 CompressedIndexedVCF

.vcf.gz with .vcf.gz.tbi file

Secondary Files

- `.tbi`

Note: For more information, visit Secondary / Accessory Files

Quickstart

```
from janis_bioinformatics.data_types.vcf import VcfTabix

w = WorkflowBuilder("my_workflow")

w.input("input_vcftabix", VcfTabix(optional=False))

# ...other workflow steps
```

This page was automatically generated on 2020-12-22.

4.12.8 CompressedTarFile

A gzipped tarfile

Quickstart

```
from janis_unix.data_types.tarfile import TarFileGz

w = WorkflowBuilder("my_workflow")

w.input("input_tarfilegz", TarFileGz(optional=False))

# ...other workflow steps
```

This page was automatically generated on 2020-12-22.

4.12.9 CompressedVCF

.vcf.gz

Quickstart

```
from janis_bioinformatics.data_types.vcf import CompressedVcf

w = WorkflowBuilder("my_workflow")

w.input("input_compressedvcf", CompressedVcf(optional=False))

# ...other workflow steps
```

This page was automatically generated on 2020-12-22.

4.12.10 CRAI

Index of the CRAM file https://samtools.github.io/hts-specs/CRAMv3.pdf

Quickstart

```
from janis_bioinformatics.data_types.crai import Crai

w = WorkflowBuilder("my_workflow")

w.input("input_crai", Crai(optional=False))

# ...other workflow steps
```

This page was automatically generated on 2020-12-22.

4.12.11 CRAM

A binary version of a SAM file, https://samtools.github.io/hts-specs/CRAMv3.pdf

Quickstart

```
from janis_bioinformatics.data_types.cram import Cram

w = WorkflowBuilder("my_workflow")

w.input("input_cram", Cram(optional=False))

# ...other workflow steps
```

This page was automatically generated on 2020-12-22.

4.12.12 CramPair

A Cram and Crai as the secondary

Secondary Files

- `.crai`

Note: For more information, visit Secondary / Accessory Files

Quickstart

```
from janis_bioinformatics.data_types.cram import CramCrai

w = WorkflowBuilder("my_workflow")

w.input("input_cramcrai", CramCrai(optional=False))

# ...other workflow steps
```

This page was automatically generated on 2020-12-22.

4.12.13 csv

A comma separated file

Quickstart

```
from janis_unix.data_types.csv import Csv

w = WorkflowBuilder("my_workflow")

w.input("input_csv", Csv(optional=False))

# ...other workflow steps
```

This page was automatically generated on 2020-12-22.

4.12.14 Directory

A directory of files

Quickstart

```
from janis_core.types.common_data_types import Directory

w = WorkflowBuilder("my_workflow")

w.input("input_directory", Directory(optional=False))

# ...other workflow steps
```

This page was automatically generated on 2020-12-22.

4.12.15 Double

An integer

Quickstart

```
from janis_core.types.common_data_types import Double

w = WorkflowBuilder("my_workflow")

w.input("input_double", Double(optional=False))

# ...other workflow steps
```

This page was automatically generated on 2020-12-22.

4.12.16 Fasta

A local file

Quickstart

```
from janis_bioinformatics.data_types.fasta import Fasta

w = WorkflowBuilder("my_workflow")

w.input("input_fasta", Fasta(optional=False))

# ...other workflow steps
```

This page was automatically generated on 2020-12-22.

4.12.17 FastaBwa

A local file

Secondary Files

- .amb
- .ann
- .bwt
- .pac
- .sa

Note: For more information, visit Secondary / Accessory Files

Quickstart

```
from janis_bioinformatics.data_types.fasta import FastaBwa

w = WorkflowBuilder("my_workflow")

w.input("input_fastabwa", FastaBwa(optional=False))

# ...other workflow steps
```

This page was automatically generated on 2020-12-22.

4.12.18 FastaFai

A local file

Secondary Files

- .fai

Note: For more information, visit Secondary / Accessory Files

Quickstart

```
from janis_bioinformatics.data_types.fasta import FastaFai

w = WorkflowBuilder("my_workflow")

w.input("input_fastafai", FastaFai(optional=False))

# ...other workflow steps
```

This page was automatically generated on 2020-12-22.

4.12.19 FastaGz

A local file

Quickstart

```
from janis_bioinformatics.data_types.fasta import FastaGz

w = WorkflowBuilder("my_workflow")

w.input("input_fastagz", FastaGz(optional=False))

# ...other workflow steps
```

This page was automatically generated on 2020-12-22.

4.12.20 FastaGzBwa

A local file

Secondary Files

- `.amb`
- `.ann`
- `.bwt`
- `.pac`
- `.sa`

Note: For more information, visit Secondary / Accessory Files

Quickstart

```
from janis_bioinformatics.data_types.fasta import FastaGzBwa

w = WorkflowBuilder("my_workflow")

w.input("input_fastagzbwa", FastaGzBwa(optional=False))

# ...other workflow steps
```

This page was automatically generated on 2020-12-22.

4.12.21 FastaGzFai

A local file

Secondary Files

- .fai

Note: For more information, visit Secondary / Accessory Files

Quickstart

```
from janis_bioinformatics.data_types.fasta import FastaGzFai

w = WorkflowBuilder("my_workflow")

w.input("input_fastagzfai", FastaGzFai(optional=False))

# ...other workflow steps
```

This page was automatically generated on 2020-12-22.

4.12.22 FastaGzWithIndexes

A local file

Secondary Files

- .amb
- .ann
- .bwt
- .pac
- .sa
- .fai
- ^.dict

Note: For more information, visit Secondary / Accessory Files

Quickstart

```
from janis_bioinformatics.data_types.fasta import FastaGzWithIndexes

w = WorkflowBuilder("my_workflow")

w.input("input_fastagzwithindexes", FastaGzWithIndexes(optional=False))

# ...other workflow steps
```

This page was automatically generated on 2020-12-22.

4.12.23 FastaWithIndexes

A local file

Secondary Files

- .fai
- .amb
- .ann
- .bwt
- .pac
- .sa
- ^.dict

Note: For more information, visit Secondary / Accessory Files

Quickstart

```
from janis_bioinformatics.data_types.fasta import FastaWithIndexes

w = WorkflowBuilder("my_workflow")

w.input("input_fastawithindexes", FastaWithIndexes(optional=False))

# ...other workflow steps
```

This page was automatically generated on 2020-12-22.

4.12.24 FastDict

A local file

Secondary Files

- ^.dict

Note: For more information, visit Secondary / Accessory Files

Quickstart

```
from janis_bioinformatics.data_types.fasta import FastaDict

w = WorkflowBuilder("my_workflow")

w.input("input_fastadict", FastaDict(optional=False))

# ...other workflow steps
```

This page was automatically generated on 2020-12-22.

4.12.25 FastGzDict

A local file

Secondary Files

- `.fai`
- `^.dict`

Note: For more information, visit Secondary / Accessory Files

Quickstart

```
from janis_bioinformatics.data_types.fasta import FastaGzDict

w = WorkflowBuilder("my_workflow")

w.input("input_fastagzdict", FastaGzDict(optional=False))

# ...other workflow steps
```

This page was automatically generated on 2020-12-22.

4.12.26 Fastq

FASTQ files are text files containing sequence data with quality score, there are different typeswith no standard: https://www.drive5.com/usearch/manual/fastq_files.html

Quickstart

```
from janis_bioinformatics.data_types.fastq import Fastq

w = WorkflowBuilder("my_workflow")

w.input("input_fastq", Fastq(optional=False))

# ...other workflow steps
```

This page was automatically generated on 2020-12-22.

4.12.27 FastqGz

FastqGz files are compressed sequence data with quality score, there are different typeswith no standard: https://en.wikipedia.org/wiki/FASTQ_format

Quickstart

```
from janis_bioinformatics.data_types.fastq import FastqGz

w = WorkflowBuilder("my_workflow")

w.input("input_fastqgz", FastqGz(optional=False))

# ...other workflow steps
```

This page was automatically generated on 2020-12-22.

4.12.28 File

A local file

Quickstart

```
from janis_core.types.common_data_types import File

w = WorkflowBuilder("my_workflow")

w.input("input_file", File(optional=False))

# ...other workflow steps
```

This page was automatically generated on 2020-12-22.

4.12.29 Filename

This class is a placeholder for generated filenames, by default it is optional and CAN be overridden, however the program has been structured in a way such that these names will be generated based on the step label. These should only be used when the tool _requires_ a filename to output and you aren't concerned what the filename should be. The Filename DataType should NOT be used as an output.

Quickstart

```
from janis_core.types.common_data_types import Filename

w = WorkflowBuilder("my_workflow")

w.input("input_filename", Filename(optional=False))

# ...other workflow steps
```

This page was automatically generated on 2020-12-22.

4.12.30 Float

A float

Quickstart

```
from janis_core.types.common_data_types import Float

w = WorkflowBuilder("my_workflow")

w.input("input_float", Float(optional=False))

# ...other workflow steps
```

This page was automatically generated on 2020-12-22.

4.12.31 Gzip

A gzipped file

Quickstart

```
from janis_unix.data_types.gunzipped import Gunzipped

w = WorkflowBuilder("my_workflow")

w.input("input_gunzipped", Gunzipped(optional=False))

# ...other workflow steps
```

This page was automatically generated on 2020-12-22.

4.12.32 HtmlFile

A HTML file

Quickstart

```
from janis_unix.data_types.html import HtmlFile

w = WorkflowBuilder("my_workflow")

w.input("input_htmlfile", HtmlFile(optional=False))

# ...other workflow steps
```

This page was automatically generated on 2020-12-22.

4.12.33 IndexedBam

A Bam and bai as the secondary

Secondary Files

- .bai

Note: For more information, visit Secondary / Accessory Files

Quickstart

```
from janis_bioinformatics.data_types.bam import BamBai

w = WorkflowBuilder("my_workflow")

w.input("input_bambai", BamBai(optional=False))

# ...other workflow steps
```

This page was automatically generated on 2020-12-22.

4.12.34 IndexedVCF

Variant Call Format:

> The Variant Call Format (VCF) specifies the format of a text file used in bioinformatics for storing gene sequence variations.
>
> Documentation: https://samtools.github.io/hts-specs/VCFv4.3.pdf

Secondary Files

- .idx

Note: For more information, visit Secondary / Accessory Files

Quickstart

```
from janis_bioinformatics.data_types.vcf import VcfIdx

w = WorkflowBuilder("my_workflow")

w.input("input_vcfidx", VcfIdx(optional=False))

# ...other workflow steps
```

This page was automatically generated on 2020-12-22.

4.12.35 Integer

An integer

Quickstart

```
from janis_core.types.common_data_types import Int

w = WorkflowBuilder("my_workflow")

w.input("input_int", Int(optional=False))

# ...other workflow steps
```

This page was automatically generated on 2020-12-22.

4.12.36 jsonFile

A JSON file file

Quickstart

```
from janis_unix.data_types.json import JsonFile

w = WorkflowBuilder("my_workflow")

w.input("input_jsonfile", JsonFile(optional=False))

# ...other workflow steps
```

This page was automatically generated on 2020-12-22.

4.12.37 KallistoIdx

A local file

Quickstart

```
from janis_bioinformatics.tools.kallisto.data_types import KallistoIdx

w = WorkflowBuilder("my_workflow")

w.input("input_kallistoidx", KallistoIdx(optional=False))

# ...other workflow steps
```

This page was automatically generated on 2020-12-22.

4.12.38 SAM

Tab-delimited text file that contains sequence alignment data

Quickstart

```
from janis_bioinformatics.data_types.sam import Sam

w = WorkflowBuilder("my_workflow")

w.input("input_sam", Sam(optional=False))

# ...other workflow steps
```

This page was automatically generated on 2020-12-22.

4.12.39 Stdout

A local file

Quickstart

```
from janis_core.types.common_data_types import Stdout

w = WorkflowBuilder("my_workflow")

w.input("input_stdout", Stdout(optional=False))

# ...other workflow steps
```

This page was automatically generated on 2020-12-22.

4.12.40 String

A string

Quickstart

```
from janis_core.types.common_data_types import String

w = WorkflowBuilder("my_workflow")

w.input("input_string", String(optional=False))

# ...other workflow steps
```

This page was automatically generated on 2020-12-22.

4.12.41 TarFile

A tarfile, ending with .tar

Quickstart

```
from janis_unix.data_types.tarfile import TarFile

w = WorkflowBuilder("my_workflow")

w.input("input_tarfile", TarFile(optional=False))

# ...other workflow steps
```

This page was automatically generated on 2020-12-22.

4.12.42 TextFile

A textfile, ending with .txt

Quickstart

```
from janis_unix.data_types.text import TextFile

w = WorkflowBuilder("my_workflow")

w.input("input_textfile", TextFile(optional=False))

# ...other workflow steps
```

This page was automatically generated on 2020-12-22.

4.12.43 tsv

A tab separated file

Quickstart

```
from janis_unix.data_types.tsv import Tsv

w = WorkflowBuilder("my_workflow")

w.input("input_tsv", Tsv(optional=False))

# ...other workflow steps
```

This page was automatically generated on 2020-12-22.

4.12.44 VCF

Variant Call Format:

> The Variant Call Format (VCF) specifies the format of a text file used in bioinformatics for storing gene sequence variations.
>
> Documentation: https://samtools.github.io/hts-specs/VCFv4.3.pdf

Quickstart

```
from janis_bioinformatics.data_types.vcf import Vcf

w = WorkflowBuilder("my_workflow")

w.input("input_vcf", Vcf(optional=False))

# ...other workflow steps
```

This page was automatically generated on 2020-12-22.

4.12.45 WhisperIdx

A local file

Secondary Files

- .whisper_idx.lut_long_dir
- .whisper_idx.lut_long_rc
- .whisper_idx.lut_short_dir
- .whisper_idx.lut_short_rc
- .whisper_idx.ref_seq_desc
- .whisper_idx.ref_seq_dir_pck
- .whisper_idx.ref_seq_rc_pck
- .whisper_idx.sa_dir
- .whisper_idx.sa_rc

Note: For more information, visit Secondary / Accessory Files

Quickstart

```
from janis_bioinformatics.tools.whisper.data_types import WhisperIdx

w = WorkflowBuilder("my_workflow")

w.input("input_whisperidx", WhisperIdx(optional=False))

# ...other workflow steps
```

This page was automatically generated on 2020-12-22.

4.12.46 Zip

A zip archive, ending with .zip

Quickstart

```
from janis_unix.data_types.zipfile import ZipFile

w = WorkflowBuilder("my_workflow")

w.input("input_zipfile", ZipFile(optional=False))

# ...other workflow steps
```

This page was automatically generated on 2020-12-22.

This page was auto-generated on 22/12/2020. Please do not directly alter the contents of this page.

4.13 Templates

These templates are used to configure Cromwell / CWLTool broadly. For more information, visit Configuring Janis.

List of templates for `janis-assistant`:

4.13.1 Local

Template ID: `local`

Quickstart

Take note below how to configure the template. This quickstart only includes the fields you absolutely require. This will write a new configuration to `~/.janis.conf`. See configuring janis for more information.

```
janis init local

# or to find out more information
janis init local --help
```

OR you can insert the following lines into your template:

```
template:
  id: local
```

Fields

Optional

ID	Type	Default	Documentation

4.13.2 Molpath

Template ID: `molpath`

Template for Peter Mac molpath, the primary difference being the initial detatched submission *sbatch* is changed to *. /etc/profile.d/modules.sh; sbatch* to handle the pipeline server.

Quickstart

Take note below how to configure the template. This quickstart only includes the fields you absolutely require. This will write a new configuration to `~/.janis.conf`. See configuring janis for more information.

```
janis init molpath

# or to find out more information
janis init molpath --help
```

OR you can insert the following lines into your template:

```
template:
  id: molpath
```

Fields

Optional

ID	Type	Default	Documentation
intermediate_execution_dir	<class 'str'>		Computation directory
container_dir	<class 'str'>	/config/binaries/singularity/containers_devel/janis/	[OPTIONAL] Override the directory singularity containers are stored in
queues	typing.Union[str, typing.List[str]]	prod_med,prod	The queue to submit jobs to
singularity_version	<class 'str'>	3.4.0	The version of Singularity to use on the cluster
send_job_emails	<class 'bool'>	False	Send Slurm job notifications using the provided email
catch_slurm_errors	<class 'bool'>	True	Fail the task if Slurm kills the job (eg: memory / time)
singularity_build_instructions	<class 'str'>		Sensible default for PeterMac template
max_cores	<class 'int'>	40	Override maximum number of cores (default: 32)
max_ram	<class 'int'>	256	Override maximum ram (default 508 [GB])
max_workflow_time	<class 'int'>	20100	The walltime of the submitted workflow "brain"
janis_memory_mb	<class 'int'>		
email_format	<class 'str'>		(null, "molpath")
log_janis_job_id_to_stdout	<class 'bool'>	False	This is already logged to STDERR, but you can also log the "Submitted batch job d" to stdout with this option set to true.
submission_sbatch	<class 'str'>	. /etc/profile.d/modules.sh; sbatch	Override the 'sbatch' command for the initial submission of janis to a compute node.

4.13.3 Pawsey

Template ID: `pawsey`

https://support.pawsey.org.au/documentation/display/US/Queue+Policies+and+Limits

Template for Pawsey. This submits Janis to the longq cluster. There is currently NO support for workflows that run for longer than 4 days, though workflows can be resubmitted after this job dies.

It's proposed that Janis assistant could resubmit itself

Quickstart

Take note below how to configure the template. This quickstart only includes the fields you absolutely require. This will write a new configuration to `~/.janis.conf`. See configuring janis for more information.

```
janis init pawsey \
    --container_dir <value>

# or to find out more information
janis init pawsey --help
```

OR you can insert the following lines into your template:

```
template:
  id: pawsey
  container_dir: <value>
```

Fields

Required

ID	Type	Documentation
container_dir	<class 'str'>	Location where to save and execute containers from

Optional

ID	Type	Default	Documentation
intermediate_execution_dir	<class 'str'>		
queues	typing.Union[str, typing.List[str]]	workq	A single or list of queues that woork should be submitted to
singularity_version	<class 'str'>	3.3.0	Version of singularity to load
catch_slurm_errors	<class 'bool'>	True	Catch Slurm errors (like OOM or walltime)
send_job_emails	<class 'bool'>	True	(requires JanisConfiguration.notifications.email to be set) Send emails for mail types END
singularity_build_instructions	<class 'str'>	singularity pull $image docker://${docker}	Instructions for building singularity, it's recommended to not touch this setting.
max_cores	<class 'int'>	28	Maximum number of cores a task can request
max_ram	<class 'int'>	128	Maximum amount of ram (GB) that a task can request
submission_queue	<class 'str'>	longq	Queue to submit the janis 'brain' to
max_workflow_time	<class 'int'>	5700	
janis_memory_mb			

4.13.4 Pbs Singularity

Template ID: `pbs_singularity`

Quickstart

Take note below how to configure the template. This quickstart only includes the fields you absolutely require. This will write a new configuration to `~/.janis.conf`. See configuring janis for more information.

```
janis init pbs_singularity

# or to find out more information
janis init pbs_singularity --help
```

OR you can insert the following lines into your template:

```
template:
  id: pbs_singularity
```

Fields

Optional

ID	Type	Default	Documentation
container_dir	<class 'str'>		Location where to save and execute containers to, this will also look at the env variables 'CWL_SINGULARITY_CACHE', 'SINGULARITY_TMPDIR'
intermediate_execution_dir	<class 'str'>		
queues	typing.Union[str, typing.List[str]]		A queue that work should be submitted to
mail_program			Mail program to pipe email to, eg: 'sendmail -t'
send_job_emails	<class 'bool'>	False	(requires JanisConfiguration.notifications.email to be set) Send emails for mail types END
catch_pbs_errors	<class 'bool'>	True	
build_instructions	<class 'str'>	singularity pull $image docker://${docker}	Instructions for building singularity, it's recommended to not touch this setting.
singularity_load_instructions			Ensure singularity with this command executed in shell
max_cores			Maximum number of cores a task can request
max_ram			Maximum amount of ram (GB) that a task can request
max_duration			Maximum amount of time in seconds (s) that a task can request
can_run_in_foreground	<class 'bool'>	True	
run_in_background	<class 'bool'>	False	

4.13.5 Pmac

Template ID: pmac

Quickstart

Take note below how to configure the template. This quickstart only includes the fields you absolutely require. This will write a new configuration to ~/.janis.conf. See configuring janis for more information.

```
janis init pmac

# or to find out more information
janis init pmac --help
```

OR you can insert the following lines into your template:

```
template:
  id: pmac
```

Fields

Optional

ID	Type	Default	Documentation
intermediate_execution_dir	<class 'str'>		Computation directory
container_dir	<class 'str'>	/config/binaries/singularity/containers_devel/janis/	[OPTIONAL] Override the directory singularity containers are stored in
queues	typing.Union[str, typing.List[str]]	prod_med,prod	The queue to submit jobs to
singularity_version	<class 'str'>	3.4.0	The version of Singularity to use on the cluster
send_job_emails	<class 'bool'>	False	Send Slurm job notifications using the provided email
catch_slurm_errors	<class 'bool'>	True	Fail the task if Slurm kills the job (eg: memory / time)
singularity_build_instructions	<class 'str'>		Sensible default for PeterMac template
max_cores	<class 'int'>	40	Override maximum number of cores (default: 32)
max_ram	<class 'int'>	256	Override maximum ram (default 508 [GB])
max_workflow_time	<class 'int'>	20100	The walltime of the submitted workflow "brain"
janis_memory_mb	<class 'int'>		
email_format	<class 'str'>		(null, "molpath")
log_janis_job_id_to_stdout	<class 'bool'>	False	This is already logged to STDERR, but you can also log the "Submitted batch job d" to stdout with this option set to true.
submission_sbatch	<class 'str'>	sbatch	
submission_node	typing.Union[str, NoneType]	papr-expanded02,	Request a specific node with '–nodelist <nodename>'

4.13.6 Singularity

Template ID: singularity

Quickstart

Take note below how to configure the template. This quickstart only includes the fields you absolutely require. This will write a new configuration to ~/.janis.conf. See configuring janis for more information.

```
janis init singularity \
    --container_dir <value>

# or to find out more information
janis init singularity --help
```

OR you can insert the following lines into your template:

```
template:
  id: singularity
  container_dir: <value>
```

Fields

Required

ID	Type	Documentation
container_dir	<class 'type'>	

Optional

ID	Type	Default	Documentation
singularity_load_instructions			Ensure singularity with this command executed in shell
container_build_instructions	<class 'str'>	singularity pull $image docker://${docker}	Instructions for building singularity, it's recommended to not touch this setting.
mail_program	<class 'str'>		Mail program to pipe email to, eg: 'sendmail -t'

4.13.7 Slurm Singularity

Template ID: `slurm_singularity`

Quickstart

Take note below how to configure the template. This quickstart only includes the fields you absolutely require. This will write a new configuration to `~/.janis.conf`. See configuring janis for more information.

```
janis init slurm_singularity

# or to find out more information
janis init slurm_singularity --help
```

OR you can insert the following lines into your template:

```
template:
  id: slurm_singularity
```

Fields

Optional

ID	Type	Default	Documentation
container_dir	<class 'str'>		Location where to save and execute containers to, this will also look at the env variables 'CWL_SINGULARITY_CACHE', 'SINGULARITY_TMPDIR'
intermediate_execution_dir	<class 'str'>		
queues	typing.Union[str, typing.List[str]]		A single or list of queues that work should be submitted to
mail_program			Mail program to pipe email to, eg: 'sendmail -t'
send_job_emails	<class 'bool'>	False	(requires JanisConfiguration.notifications.email to be set) Send emails for mail types END
catch_slurm_errors	<class 'bool'>	True	Catch Slurm errors (like OOM or walltime)
build_instructions	<class 'str'>	singularity pull $image docker://${docker}	Instructions for building singularity, it's recommended to not touch this setting.
singularity_load_instructions			Ensure singularity with this command executed in shell
max_cores			Maximum number of cores a task can request
max_ram			Maximum amount of ram (GB) that a task can request
max_duration			Maximum amount of time in seconds (s) that a task can request
can_run_in_foreground	<class 'bool'>	True	
run_in_background	<class 'bool'>	False	
sbatch	<class 'str'>	sbatch	Override the sbatch command

4.13.8 Spartan

Template ID: `spartan`

https://dashboard.hpc.unimelb.edu.au/

Quickstart

Take note below how to configure the template. This quickstart only includes the fields you absolutely require. This will write a new configuration to `~/.janis.conf`. See configuring janis for more information.

```
janis init spartan \
    --container_dir <value>

# or to find out more information
janis init spartan --help
```

OR you can insert the following lines into your template:

```
template:
  id: spartan
  container_dir: <value>
```

Fields

Required

ID	Type	Documentation
container_dir	<class 'str'>	

Optional

ID	Type	Default	Documentation
intermediate_execution_dir	<class 'str'>		computation directory for intermediate files (defaults to <exec>/execution OR <outputdir>/janis/execution)
queues	typing.Union[str, typing.List[str]]	physical	The queue to submit jobs to
singularity_version	<class 'str'>	3.5.3	
send_job_emails	<class 'bool'>	True	Send SLURM job emails to the listed email address
catch_slurm_errors	<class 'bool'>	True	Fail the task if Slurm kills the job (eg: memory / time)
max_cores	<class 'int'>	32	Override maximum number of cores (default: 32)
max_ram	<class 'int'>	508	Override maximum ram (default 508 [GB])
submission_queue	<class 'str'>	physical	
max_workflow_time	<class 'int'>	20100	
janis_memory_mb			

4.13.9 Wehi

Template ID: `wehi`

Quickstart

Take note below how to configure the template. This quickstart only includes the fields you absolutely require. This will write a new configuration to `~/.janis.conf`. See configuring janis for more information.

```
janis init wehi \
    --container_dir <value>

# or to find out more information
janis init wehi --help
```

OR you can insert the following lines into your template:

```
template:
  id: wehi
  container_dir: <value>
```

Fields

Required

ID	Type	Documentation
container_dir	<class 'str'>	

Optional

ID	Type	Default	Documentation
intermediate_execution_dir	<class 'str'>		A location where the execution should take place
queues	typing.Union[typing.List[str], str]		A single or list of queues that woork should be submitted to
singularity_version	<class 'str'>	3.4.1	Version of singularity to load
catch_pbs_errors	<class 'bool'>	True	Catch PBS errors (like OOM or walltime)
send_job_emails	<class 'bool'>	True	(requires JanisConfiguration.notifications.email to be set) Send emails for mail types END
build_instructions			Instructions for building singularity, it's recommended to not touch this setting.
max_cores	<class 'int'>	40	Maximum number of cores a task can request
max_ram	<class 'int'>	256	Maximum amount of ram (GB) that a task can request

This page was auto-generated on 22/12/2020. Please do not directly alter the contents of this page.

4.14 List of operators

This document is automatically generated, and contains the operators that Janis provides. They are split up by section (Regular operators, Logical operators, Standard library) with the relevant Python import.

For more information about expressions and operators, visit Expressions in Janis.

4.14.1 Regular operators

Import with: `from janis_core.operators.operator import *`

- **IndexOperator**: `Array[X], Int -> X`
- **AsStringOperator**: `X -> String`
- **AsBoolOperator**: `X -> Boolean`
- **AsIntOperator**: `X -> Int`
- **AsFloatOperator**: `X -> Float`

4.14.2 Logical operators

Import with: `from janis_core.operators.logical import *`

- **IsDefined**: `Any? -> Boolean`
- **If**: `(condition: Boolean, value_if_true: X, value_if_false: Y) -> Union[X,Y]`

- **NotOperator**: `Boolean -> Boolean`
- **AndOperator**: `Boolean, Boolean -> Boolean`
- **OrOperator**: `Boolean, Boolean -> Boolean`
- **EqualityOperator**: `Any, Any -> Boolean`
- **InequalityOperator**: `Any, Any -> Boolean`
- **GtOperator**: `Comparable, Comparable -> Boolean`
- **GteOperator**: `Comparable, Comparable -> Boolean`
- **LtOperator**: `Comparable, Comparable -> Boolean`
- **LteOperator**: `Comparable, Comparable -> Boolean`
- **AddOperator**: `Any, Any -> Any`
- **SubtractOperator**: `NumericType, NumericType -> NumericType`
- **MultiplyOperator**: `Numeric, NumericType -> NumericType`
- **DivideOperator**: `Numeric, NumericType -> NumericType`
- **FloorOperator**: `Numeric, NumericType -> Int`
- **CeilOperator**: `Numeric, NumericType -> Int`
- **RoundOperator**: `Numeric, NumericType -> Int`

4.14.3 Standard library

Import with: `from janis_core.operators.standard import *`

- **ReadContents**: `File -> String`
- **JoinOperator**: `Array[X], String -> String`
- **BasenameOperator**: `Union[File, Directory] -> String`
- **TransformOperator**: `Array[Array[X]] -> Array[Array[x]]`
- **LengthOperator**: `Array[X] -> Int`
- **FlattenOperator**: `Array[Array[X]] -> Array[x]`
- **ApplyPrefixOperator**: `String, Array[String] -> String`
- **FileSizeOperator**: `File -> Float`

  Returned in MB: Note that this does NOT include the reference files (yet)

- **FirstOperator**: `Array[X?] -> X`
- **FilterNullOperator**: `Array[X?] -> Array[X]`

This page was autogenerated.

4.15 Janis Shed and Toolboxes

One important component of Janis is the Toolbox, a set of command tools and workflows that are immediately available to you. There are toolboxes for each domain, and we refer to the collection of these toolboxes as a “JanisShed”.

In essence, they're Python packages that Janis knows about and can use, currently there are two toolboxes:

- Unix (GitHub | Docs)
- Bioinformatics (GitHub | Docs)

4.15.1 Adding a tool

To add a tool to an existing toolbox, please refer to Adding a tool to a toolbox.

4.15.2 Custom Toolbox

You might have a set of tools that you you want to use in Janis (and take advantage of the other features), but may not be allowed to publish these (and they're container). In this case, we'd recommend you set up your own toolbox as a Python module, and by setting a few entrypoints Janis will be able to pick up your tools.

We won't discuss how to setup a Python package from scratch, but you might want to follow the existing toolbox repos as a guide. Janis uses entrypoints to find existing toolboxes, here's a great guide on Python Entry Points Explained.

Specifically, you'll need to add the lines to your `setup.py` (replacing `bioinformatics` with a keyword to describe your extension, and `janis_bioinformatics` with your package name):

```
entry_points={
    "janis.extension": ["bioinformatics=janis_bioinformatics"],
    "janis.tools": ["bioinformatics=janis_bioinformatics.tools"],
    "janis.types": ["bioinformatics=janis_bioinformatics.data_types"],
},
```

Currently, the Janis assistant doesn't support configuring Cromwell or CWLTool to interact with private container repositories. If this is something you need, I'd recommend reaching out on GitHub and Gitter, or translating your output to run the workflow on your own engine.

4.15.3 Development Decisions

The design of the Janis tool repositories is a little strange, but we'll discuss some of the design decisions we made to give context.

We know that the underling command line for tools is pretty constant between versions, and we didn't want to create a new tool definition for each version for a few reasons:

- Easy to fix errors if code is in one place
- Makes pull requests easy when there's minimal code changes

For groups of tools that might share common inputs (eg: `GATK4`), we wanted to declare a Gatk4Base, where we can limit the memory of each GATK tool by inserting the parameter `--java-options '-Xmx{memory}G'`. Doing this in the GATK base means it's implemented in one location and ALL of the GATK tools get this functionality for free.

For these reasons, janis-bioinformatics uses a class inheritance structure, but this sometimes makes for some awkward imports.

1. Often the tool provider will declare some base, with some handy methods. For example, our Samtools base command is `["samtools", SamtoolsCommand]`, so we'll create an abstract method for `samtools_command` that each tool must implement, and we can take care of the base command for them. We'll also declare a version of the base GATK with the version and containers method filled, we'll see why in a second.

```
import abc

class SamToolsBase(abc.ABC):
    @abc.abstractmethod
    def samtools_command(self):
        pass

    def base_command(self):
        return ["samtools", self.samtools_command()]

    # other handy methods

class SamTools_1_9:
    def container(self):
        return "quay.io/biocontainers/samtools:1.9--h8571acd_11"

    def version(self):
        return "1.9.0"
```

2. Now we can create our base definition for our class. Although a CommandTool usually requires a container and version, we'll use the one we declared earlier in the next step. Let's create a rough definition for Samtools Flagstat

```
class SamToolsFlagstatBase(SamToolsToolBase, ABC):
    def samtools_command(self):
        return "flagstat"

    def inputs(self):
        return [
            ToolInput("bam", Bam(), position=10)
        ]

    def outputs(self):
        return [ToolOutput("out", Stdout(TextFile))]
```

3. We can combine our Samtools version and FlagstatBase to create a version of flagstat, eg:

```
class SamToolsFlagstat_1_9(SamTools_1_9, SamToolsFlagstatBase):
    pass
```

Suggestions

See this GitHub issue for suggestions to improve the structure: GitHub: janis-bioinformatics/issues/92

4.16 Framework Documentation

There are two major components to constructing workflows, building tools and assembling workflows.

4.16.1 Workflow

Manages the connections between tools

Declaration

There are two major ways to construct a workflow:

- Inline using the `janis.WorkflowBuilder` class,
- or Inheriting from the `janis.Workflow` class and implementing the required methods.

class janis.**WorkflowBuilder**(*identifier: str = None, friendly_name: str = None, version: str = None, metadata: janis_core.utils.metadata.WorkflowMetadata = None, tool_provider: str = None, tool_module: str = None, doc: str = None*)

class janis.**Workflow**(***connections*)

Advanced Workflows

Janis allows you to dynamically create workflows based on inputs. More information can be found on the Dynamic Workflows page.

Overview

The `janis.Workflow` and `janis.WorkflowBuilder` classes exposes inputs, and manages the connections between these inputs, tools and exposes some outputs.

A `janis.WorkflowBuilder` is the class used inline to declare workflows. The `janis.Workflow` class should only be inherited through subclasses.

A workflow does not directly execute, but declares what inputs a *janis.CommandTool* should receive.

A representation of a workflow can be exported to `cwl` or `wdl` through the **:method:‘janis.Workflow.translate()‘** function.

Translating

Currently Janis supports two translation targets:

1. Common Workflow Language
2. Workflow Description Language

Workflow.**translate**(*translation: Union[str, janis_core.translationdeps.supportedtranslations.SupportedTranslation], to_console=True, tool_to_console=False, to_disk=False, write_inputs_file=True, with_docker=True, with_hints=False, with_resource_overrides=False, validate=False, should_zip=True, export_path='./{name}', merge_resources=False, hints=None, allow_null_if_not_optional=True, additional_inputs: Dict[KT, VT] = None, max_cores=None, max_mem=None, max_duration=None, allow_empty_container=False, container_override: dict = None*)

Structure of a workflow

A workflow has the following _nodes_:

- Inputs - *janis.Workflow.input()*
- Steps - *janis.Workflow.step()*
- Outputs - *janis.Workflow.output()*

Once an node has been added to the workflow, it may be referenced through dot-notation on the workflow. For this reason, identifiers have certain naming restrictions. In the following examples we're going to create an *inline* workflow using the `WorkflowBuilder` class.

Creating an input

An input requires a unique identifier (string) and a `janis.DataType`.

`Workflow.`**`input`**(*identifier: str, datatype: Union[Type[Union[str, float, int, bool]], janis_core.types.data_types.DataType, Type[janis_core.types.data_types.DataType]], default: any = None, value: any = None, doc: Union[str, janis_core.tool.documentation.InputDocumentation, Dict[str, any]] = None*)

Create an input node on a workflow :return:

The input node is returned from this function, and is also available as a property on a workflow (accessible through dot-notation OR index notation).

```
import janis as j

w = j.WorkflowBuilder("myworkflow")
myInput = w.input("myInput", String)
myInput == w.myInput == w["myInput"] # True
```

Note: Default vs Value: The input

Creating a step

A step requires a unique identifier (string), a mapped tool (either a *janis.CommandTool* or *janis.Workflow* called with it's inputs), scattering information (if required).

`Workflow.`**`step`**(*identifier: str, tool: janis_core.tool.tool.Tool, scatter: Union[str, List[str], janis_core.utils.scatter.ScatterDescription] = None, _foreach: Union[janis_core.operators.selectors.Selector, List[janis_core.operators.selectors.Selector]] = None, when: Optional[janis_core.operators.operator.Operator] = None, ignore_missing=False, doc: str = None*)

Construct a step on this workflow.

Parameters

- **`identifier`** – The identifier of the step, unique within the workflow.
- **`tool`** – The tool that should run for this step.
- **`scatter`** (*Union[str, ScatterDescription]*) – Indicate whether a scatter should occur, on what, and how.

- **when** (*Optional[Operator]*) – An operator / condition that determines whether the step should run
- **ignore_missing** – Don't throw an error if required params are missing from this function
- **_foreach** – NB: this is unimplemented. Iterate for each value of this resolves list, where you should use the "ForEachSelector" to select each value in this iterable.

Returns

Janis will throw an error if all the *required* inputs are not provided. You can provide the parameter `ignore_missing=True` to the step function to skip this check.

```
from janis.unix.tools.echo import Echo

# Echo has the required input: "inp": String
# https://janis.readthedocs.io/en/latest/tools/unix/echo.html

echoStep = w.step("echoStep", Echo(inp=w.myInput))
echoStep == w.echoStep == w["echoStep"] # True
```

Creating an output

An output requires a unique identifier (string), an output source and an *optional* `janis.DataType`. If a data type is provided, it is type-checked against the output source. Don't be put off by the automatically generated interface for the output method, it's there to be exhaustive for the type definitions.

Here is the (simplified) method definition:

```
def output(
    self,
    identifier: str,
    datatype: Optional[ParseableType] = None,
    source: Union[Selector, ConnectionSource]=None # or List[Selector,␣
↪ConnectionSource]
    output_folder: Union[str, Selector, List[Union[str, Selector]]] = None,
    output_name: Union[bool, str, Selector, ConnectionSource] = True, # let janis␣
↪decide output name
    extension: Optional[str] = None, # file extension if janis names file
    doc: Union[str, OutputDocumentation] = None,
):
```

`Workflow.`**`output`**(*identifier: str, datatype: Union[Type[Union[str, float, int, bool]], janis_core.types.data_types.DataType, Type[janis_core.types.data_types.DataType], None] = None, source: Union[List[Union[janis_core.operators.selectors.Selector, janis_core.graph.node.Node, janis_core.operators.selectors.StepOutputSelector, Tuple[janis_core.graph.node.Node, str]]], janis_core.operators.selectors.Selector, janis_core.graph.node.Node, janis_core.operators.selectors.StepOutputSelector, Tuple[janis_core.graph.node.Node, str]] = None, output_folder: Union[str, janis_core.operators.selectors.Selector, List[Union[janis_core.operators.selectors.Selector, str]]] = None, output_name: Union[bool, str, janis_core.operators.selectors.Selector, janis_core.graph.node.Node, janis_core.operators.selectors.StepOutputSelector, Tuple[janis_core.graph.node.Node, str]] = True, extension: Optional[str] = None, doc: Union[str, janis_core.tool.documentation.OutputDocumentation] = None*)

Create an output on a workflow

Parameters

- **`identifier`** – The identifier for the output
- **`datatype`** – Optional data type of the output to check. This will be automatically inferred if not provided.
- **`source`** – The source of the output, must be an output to a step node
- **`output_folder`** – Decides the output folder(s) where the output will reside. If a list is passed, it represents a structure of nested directories, the first element being the root directory.
  - None (default): the assistant will copy to the root of the output directory
  - Type[Selector]: will be resolved before the workflow is run, this means it may only depend on the inputs

  NB: If the output_source is an array, a “shard_n” will be appended to the output_name UNLESS the output_source also resolves to an array, which the assistant can unwrap multiple dimensions of arrays ONLY if the number of elements in the output_scattered source and the number of resolved elements is equal.
- **`output_name`** –

  Decides the name of the output (without extension) that an output will have:
  - True (default): the assistant will choose an output name based on output identifier (tag),
  - None / False: the assistant will use the original filename (this might cause filename conflicts)
  - Type[Selector]: will be resolved before the workflow is run, this means it may only depend on the inputs

  NB: If the output_source is an array, a “shard_n” will be appended to the output_name UNLESS the output
  also resolves to an array, which the assistant can unwrap multiple dimensions of arrays.
- **`extension`** – The extension to use if janis renames the output. By default, it will pull the extension from the inherited data type (eg: CSV -> “.csv”), or it will attempt to pull the extension from the file.

Returns janis.WorkflowOutputNode

You are unable to connect an input node directly to an output node, and an output node cannot be referenced as a step input.

```
# w.echoStep added to workflow
w.output("out", source=w.echoStep)
```

Subclassing Workflow

Instead of creating inline workflows, it’s possible to subclass *`janis.Workflow`*, implement the required methods which allows a tool to have documentation automatically generated.

Required methods:

`Workflow.`**`id`**`()` → str

`Workflow.`**`friendly_name`**`()`

Overriding this method is not required UNLESS you distribute your tool. Generating the docs will fail if your tool does not provide a name.

Returns A friendly name of your tool

`Workflow.`**`constructor`**`()`

A place to construct your workflows. This is called directly after initialisation. :return:

Within the `constructor` method, you have access to `self` to add inputs, steps and outputs.

OPTIONAL:

`Workflow.`**`bind_metadata`**`()`

A convenient place to add metadata about the tool. You are guaranteed that self.metadata will exist. It's possible to return a new instance of the ToolMetadata / WorkflowMetadata which will be rebound. This is usually called after the initialiser, though it may be called multiple times. :return:

Examples

Inline example

The Echo tool has one inputs `inp` of type string, and one output `out`.

```
import janis as j
from janis.unix.tools.echo import Echo

w = j.WorkflowBuilder("my_workflow")
w.input("my_input", String)
echoStep = w.step("echo_step", Echo(inp=w.my_input))
w.output("out", source=w.echo_step)

# Will print the CWL, input file and relevant tools to the console
w.translate("cwl", to_disk=False)  # or "wdl"
```

Subclass example

```
import janis as j
from janis.unix.tools.echo import Echo

class MyWorkflow(j.Workflow):

    def id(self):
        return "my_workflow"

    def friendly_name(self):
        return "My workflow"

    def constructor(self):
        self.input("my_input", String)
        echoStep = w.step("echo_step", Echo(inp=self.my_input))
        self.output("out", source=self.echo_step)

    # optional

    def metadata(self):
        self.metadata.author = "Michael Franklin"
```

(continues on next page)

(continued from previous page)

```
        self.metadata.version = "v1.0.0"
        self.metadata.documentation = "A tool that echos the input to standard_out
```

4.16.2 Command Tool

A class that wraps a CommandLineTool with named argument

Declaration

class janis.**CommandTool**(***connections*)

A CommandTool is an interface between Janis and a program to be executed. Simply put, a CommandTool has a name, a command, inputs, outputs and a container to run in.

This class can be inherited to created a CommandTool, else a CommandToolBuilder may be used.

class janis.**CommandToolBuilder**(*tool: str, base_command: Union[str, List[str], None], inputs: List[janis_core.tool.commandtool.ToolInput], outputs: List[janis_core.tool.commandtool.ToolOutput], container: str, version: str, friendly_name: Optional[str] = None, arguments: List[janis_core.tool.commandtool.ToolArgument] = None, env_vars: Dict[KT, VT] = None, tool_module: str = None, tool_provider: str = None, metadata: janis_core.utils.metadata.ToolMetadata = None, cpus: Union[int, Callable[[Dict[str, Any]], int]] = None, memory: Union[int, Callable[[Dict[str, Any]], int]] = None, time: Union[int, Callable[[Dict[str, Any]], int]] = None, disk: Union[int, Callable[[Dict[str, Any]], int]] = None, directories_to_create: Union[janis_core.operators.selectors.Selector, str, List[Union[janis_core.operators.selectors.Selector, str]]] = None, files_to_create: Union[Dict[str, Union[str, janis_core.operators.selectors.Selector]], List[Tuple[Union[janis_core.operators.selectors.Selector, str], Union[janis_core.operators.selectors.Selector, str]]]] = None, doc: str = None*)

__init__(*tool: str, base_command: Union[str, List[str], None], inputs: List[janis_core.tool.commandtool.ToolInput], outputs: List[janis_core.tool.commandtool.ToolOutput], container: str, version: str, friendly_name: Optional[str] = None, arguments: List[janis_core.tool.commandtool.ToolArgument] = None, env_vars: Dict[KT, VT] = None, tool_module: str = None, tool_provider: str = None, metadata: janis_core.utils.metadata.ToolMetadata = None, cpus: Union[int, Callable[[Dict[str, Any]], int]] = None, memory: Union[int, Callable[[Dict[str, Any]], int]] = None, time: Union[int, Callable[[Dict[str, Any]], int]] = None, disk: Union[int, Callable[[Dict[str, Any]], int]] = None, directories_to_create: Union[janis_core.operators.selectors.Selector, str, List[Union[janis_core.operators.selectors.Selector, str]]] = None, files_to_create: Union[Dict[str, Union[str, janis_core.operators.selectors.Selector]], List[Tuple[Union[janis_core.operators.selectors.Selector, str], Union[janis_core.operators.selectors.Selector, str]]]] = None, doc: str = None*)

Builder for a CommandTool.

Parameters

- **tool** – Unique identifier of the tool
- **friendly_name** – A user friendly name of your tool (must be implemented for generated docs)
- **base_command** – The command of the tool to execute, usually the tool name or path and not related to any inputs.
- **inputs** – A list of named tool inputs that will be used to create the command line.
- **outputs** – A list of named outputs of the tool; a `ToolOutput` declares how the output is captured.
- **arguments** – A list of arguments that will be used to create the command line.
- **container** – A link to an OCI compliant container accessible by the engine.
- **version** – Version of the tool.
- **env_vars** – A dictionary of environment variables that should be defined within the container.
- **tool_module** – Unix, bioinformatics, etc.
- **tool_provider** – The manafacturer of the tool, eg: Illumina, Samtools
- **metadata** – Metadata object describing the Janis tool interface
- **cpu** – An integer, or function that takes a dictionary of hints and returns an integer in 'number of CPUs'
- **memory** – An integer, or function that takes a dictionary of hints and returns an integer in 'GBs'
- **time** – An integer, or function that takes a dictionary of hints and returns an integer in 'seconds'
- **disk** – An integer, or function that takes a dictionary of hints and returns an integer in 'GBs'
- **directories_to_create** – A list of directories to create, accepts an expression (selector / operator)
- **files_to_create** – Either a List of tuples [path: Selector, contents: Selector],

or a dictionary {"path": contents}. The list of tuples allows you to use an operator for the pathname :param doc: Documentation string

Overview

A *`janis.CommandTool`* is the programmatic interface between a set of inputs, how these inputs bind on the Command Line to call the tool, and how to collect the outputs. It would be rare that you should directly instantiate `CommandTool`, instead you should subclass and and override the methods you need as declared below.

Like a workflow, there are two methods to declare a command tool:

- Use the `CommandToolBuilder` class
- Inherit from the `CommandTool` class,

Template

CommandToolBuilder:

```
import janis_core as j

ToolName = j.CommandToolBuilder(
    tool: str="toolname",
    base_command=["base", "command"],
    inputs: List[j.ToolInput]=[],
    outputs: List[j.ToolOutput]=[],
    container="container/name:version",
    version="version",
    friendly_name=None,
    arguments=None,
    env_vars=None,
    tool_module=None,
    tool_provider=None,
    metadata: ToolMetadata=j.ToolMetadata(),
    cpu: Union[int, Callable[[Dict[str, Any]], int]]=None,
    memory: Union[int, Callable[[Dict[str, Any]], int]]=None,
)
```

This is equivalent to the inherited template:

```
from typing import List, Optional, Union
import janis_core as j

class ToolName(j.CommandTool):
    def tool(self) -> str:
        return "toolname"

    def base_command(self) -> Optional[Union[str, List[str]]]:
        pass

    def inputs(self) -> List[j.ToolInput]:
        return []

    def outputs(self) -> List[j.ToolOutput]:
        return []

    def container(self) -> str:
        return ""

    def version(self) -> str:
        pass

     # optional

     def arguments(self) -> List[j.ToolArgument]:
         return []

     def env_vars(self) -> Optional[Dict[str, Union[str, Selector]]]:
         return {}

     def friendly_name(self) -> str:
         pass
```

(continues on next page)

(continued from previous page)

```
    def tool_module(self) -> str:
        pass

    def tool_provider(self) -> str:
        pass

    def cpu(self, hints: Dict) -> int:
        pass

    def memory(self, hints: Dict) -> int:
        pass

    def bind_metadata(self) -> j.ToolMetadata:
        pass
```

Structure

A new tool definition must subclass the *janis.CommandTool* class and implement the required abstract methods:

- `janis.CommandTool.tool(self) -> str`: Unique identifier of the tool
- `janis.CommandTool.base_command(self) -> str`: The command of the tool to execute, usually the tool name or path and not related to any inputs.
- `janis.CommandTool.inputs(self) -> List[`*`janis.ToolInput`*`]`: A list of named tool inputs that will be used to create the command line.
- `janis.CommandTool.arguments(self) -> List[`*`janis.ToolArgument`*`]`: A list of arguments that will be used to create the command line.
- `janis.CommandTool.outputs(self) -> List[`*`janis.ToolOutput`*`]`: A list of named outputs of the tool; a `ToolOutput` declares how the output is captured.
- `janis.CommandTool.container(self)`: A link to an OCI compliant container accessible by the engine. Previously, `docker()`.
- `janis.CommandTool.version(self)`: Version of the tool.
- `janis.CommandTool.env_vars(self) -> Optional[Dict[str, Union[str, Selector]]]`: A dictionary of environment variables that should be defined within the container.

To better categorise your tool, you can additionally implement the following methods:

- `janis.Tool.friendly_name(self)`: A user friendly name of your tool (must be implemented for generated docs)
- `janis.Tool.tool_module(self)`: Unix, bioinformatics, etc.
- `janis.Tool.tool_provider(self)`: The manufacturer of the tool, eg: Illumina, Samtools

Tool Input

class janis.**ToolInput**(*tag: str, input_type: Union[Type[Union[str, float, int, bool]], janis_core.types.data_types.DataType, Type[janis_core.types.data_types.DataType]], position: Optional[int] = None, prefix: Optional[str] = None, separate_value_from_prefix: bool = None, prefix_applies_to_all_elements: bool = None, presents_as: str = None, secondaries_present_as: Dict[str, str] = None, separator: str = None, shell_quote: bool = None, localise_file: bool = None, default: Any = None, doc: Union[str, janis_core.tool.documentation.InputDocumentation, None] = None*)

__init__(*tag: str, input_type: Union[Type[Union[str, float, int, bool]], janis_core.types.data_types.DataType, Type[janis_core.types.data_types.DataType]], position: Optional[int] = None, prefix: Optional[str] = None, separate_value_from_prefix: bool = None, prefix_applies_to_all_elements: bool = None, presents_as: str = None, secondaries_present_as: Dict[str, str] = None, separator: str = None, shell_quote: bool = None, localise_file: bool = None, default: Any = None, doc: Union[str, janis_core.tool.documentation.InputDocumentation, None] = None*)

A `ToolInput` represents an input to a tool, with parameters that allow it to be bound on the command line. The ToolInput must have either a position or prefix set to be bound onto the command line.

Parameters

- **tag** – The identifier of the input (unique to inputs and outputs of a tool)
- **input_type** (`janis.ParseableType`) – The data type that this input accepts
- **position** – The position of the input to be applied. (Default = 0, after the base_command).
- **prefix** – The prefix to be appended before the element. (By default, a space will also be applied, see `separate_value_from_prefix` for more information)
- **separate_value_from_prefix** – (Default: True) Add a space between the prefix and value when `True`.
- **prefix_applies_to_all_elements** – Applies the prefix to each element of the array (Array inputs only)
- **shell_quote** – Stops shell quotes from being applied in all circumstances, useful when joining multiple commands together.
- **separator** – The separator between each element of an array (defaults to ' ')
- **localise_file** – Ensures that the file(s) are localised into the execution directory.
- **default** – The default value to be applied if the input is not defined.
- **doc** – Documentation string for the ToolInput, this is used to generate the tool documentation and provide

hints to the user.

Note: A ToolInput (and `ToolArgument`) must have either a `position` AND / OR `prefix` to be bound onto the command line.

- The `prefix` is a string that precedes the inputted value. By default the prefix is separated by a space, however this can be removed with `separate_value_from_prefix=False`.

- The `position` represents the order of how arguments are bound onto the command line. Lower numbers get a higher priority, not providing a number will default to 0.
- `prefix_applies_to_all_elements` applies the prefix to each element in an array (only applicable for array inputs).
- The `localise_file` attribute places the file input within the execution directory.
- `presents_as` is a mechanism for overriding the name to localise to. The `localise_file` parameter MUST be set to *True* for `presents_as`
- `secondaries_present_as` is a mechanism for overriding the format of secondary files. `localise_file` does NOT need to be set for this functionality to work. In CWL, this relies on https://github.com/common-workflow-language/cwltool/pull/1233

Tool Argument

class janis.**ToolArgument**(*value: Any*, *prefix: Optional[str] = None*, *position: Optional[int] = 0*, *separate_value_from_prefix=None*, *doc: Union[str, janis_core.tool.documentation.DocumentationMeta, None] = None*, *shell_quote: bool = None*)

__init__(*value: Any*, *prefix: Optional[str] = None*, *position: Optional[int] = 0*, *separate_value_from_prefix=None*, *doc: Union[str, janis_core.tool.documentation.DocumentationMeta, None] = None*, *shell_quote: bool = None*)

A `ToolArgument` is a CLI parameter that cannot be override (at runtime). The value can

Parameters

- **value** (`str` | `janis.InputSelector` | `janis.StringFormatter`) –
- **position** – The position of the input to be applied. (Default = 0, after the base_command).
- **prefix** – The prefix to be appended before the element. (By default, a space will also be applied, see `separate_value_from_prefix` for more information)
- **separate_value_from_prefix** – (Default: True) Add a space between the prefix and value when `True`.
- **doc** – Documentation string for the argument, this is used to generate the tool documentation and provide
- **shell_quote** – Stops shell quotes from being applied in all circumstances, useful when joining multiple commands together.

Tool Output

class janis.**ToolOutput**(*tag: str, output_type: Union[Type[Union[str, float, int, bool]], janis_core.types.data_types.DataType, Type[janis_core.types.data_types.DataType]], selector: Union[janis_core.operators.selectors.Selector, str, None] = None, presents_as: str = None, secondaries_present_as: Dict[str, str] = None, doc: Union[str, janis_core.tool.documentation.OutputDocumentation, None] = None, glob: Union[janis_core.operators.selectors.Selector, str, None] = None, _skip_output_quality_check=False*)

__init__(*tag: str, output_type: Union[Type[Union[str, float, int, bool]], janis_core.types.data_types.DataType, Type[janis_core.types.data_types.DataType]], selector: Union[janis_core.operators.selectors.Selector, str, None] = None, presents_as: str = None, secondaries_present_as: Dict[str, str] = None, doc: Union[str, janis_core.tool.documentation.OutputDocumentation, None] = None, glob: Union[janis_core.operators.selectors.Selector, str, None] = None, _skip_output_quality_check=False*)

A ToolOutput instructs the the engine how to collect an output and how it may be referenced in a workflow.

Parameters

- **tag** – The identifier of a output, must be unique in the inputs and outputs.
- **output_type** – The type of output that is being collected.
- **selector** – How to collect this output, can accept any `janis.Selector`.
- **glob** – (DEPRECATED) An alias for *selector*
- **doc** – Documentation on what the output is, used to generate docs.
- **_skip_output_quality_check** – DO NOT USE THIS PARAMETER, it's a scapegoat for parsing CWL ExpressionTools when an cwl.output.json is generated

4.16.3 Python Tool

Note: Available in v0.9.0 and later

A PythonTool is a Janis mechanism for running arbitrary Python code inside a container.

class `janis.`**PythonTool**(***connections*)

static code_block(***kwargs*) → Dict[str, Any]

This code block must be 100% self contained. All libraries and functions must be imported and declared from within this block. :param kwargs: :return:

How to build my own Tool

- Create a class that inherits from `PythonTool`,
- The PythonTool uses the code signature for code_block to determine the inputs. This means you must use Python *type annotations*.
- The default image is `python:3.8.1`, this is overridable by overriding the container method.

Extra notes about the code_block:

- It must be totally self contained (including ALL imports and functions)
- The only information you have available is those that you specify within your method.
- **Annotation notes:**
  - You can annotate with regular Python types, like: `str` or `int`
  - You can annotate optionality with typings, like: `Optional[str]` or `List[Optional[str]]`
  - Do NOT use `Array(String)`, use `List[String]` instead.

  - Annotating with Janis types (like String, File, BamBai, etc) might cause issues with your type hints in an IDE.

- A File type will be presented to your function as a string. You will need to open the file within your function.
- You must return a dictionary with keys corresponding to the outputs of your tool. - File outputs should be presented as a string, and will be coerced to a File / Directory later.
- Do NOT use `j.$TYPE` annotations (prefixed with `j.`, eg: `j.File` or `j.String`) as annotations as this will fail at runtime.

```
from janis_core import PythonTool, TOutput, File
from typing import Dict, Optional, List, Any

class MyPythonTool(PythonTool):
    @staticmethod
    def code_block(in_string: str, in_file: File, in_integer: int) -> Dict[str, Any]:
        from shutil import copyfile

        copyfile(in_file, "./out.file")

        return {
            "myout": in_string + "-appended!",
            "myfileout": "out.file",
            "myinteger": in_integer + 1
        }

    def outputs(self) -> List[TOutput]:
        return [
            TOutput("myout", str),
            TOutput("myfileout", File),
            TOutput("myinteger", int)
        ]

    def id(self) -> str:
        return "MyPythonTool"

    def version(self):
        return "v0.1.0"
```

How to include a File inside your container

Use the type annotation `File`, and open it yourself, for example:

```
from janis_core import PythonTool, TOutput, File
from typing import Dict, Optional, List, Any

class CatPythonTool(PythonTool):
    @staticmethod
    def code_block(my_file: File) -> Dict[str, Any]:

        with open(my_file, "r") as f:
            return {
                "out": f.read()
            }

    def outputs(self) -> List[TOutput]:
```

(continues on next page)

(continued from previous page)

```
        return [
            TOutput("out", str)
        ]
```

How to include an Array of strings inside my container

```
from janis_core import PythonTool, TOutput, File
from typing import Dict, Optional, List, Any

class JoinStrings(PythonTool):
    @staticmethod
    def code_block(my_strings: List[str], separator: str=",") -> Dict[str, Any]:

        return {"out": separator.join(my_strings)}

    def outputs(self) -> List[TOutput]:
        return [
            TOutput("out", str)
        ]
```

4.16.4 Code Tool

Note: BETA: Available in v0.9.0 and later

A code tool is an abstract concept in Janis, that aims to execute arbitrary code inside a container.

Currently there is only one type of CodeTool:

- *janis.PythonTool*

The aim is to make it simpler to perform basic steps within your container, for example say I want to perform some basic processing of a file that isn't trivial in Janis (and hence CWL / WDL) but is in a programming language, Janis gives you the functionality to make this possible.

Creating a new code tool type

This process is designed to be fairly simple, but there are a few important notes:

- Your tool must log ONLY a JSON string to stdout (this will get parsed later)
- The `prepared_script` block must return a string that will get written to a file within the container to be executed that can accept inputs via command line. Within the PythonTool, you'll see that a parser (by `argparse`) is generated in this method.

```
import janis_core as j

class LanguageTool(CodeTool):
    # You might leave these fields to be overriden by the user
    def inputs(self) -> List[TInput]:
        pass
```

(continues on next page)

(continued from previous page)

```
    def outputs(self) -> List[TOutput]:
        pass

    def base_command(self):
        pass

    def script_name(self):
        pass

    def container(self):
        pass

    def prepared_script(self):
        pass

    def id(self) -> str:
        pass

    def version(self):
        pass
```

4.16.5 Types

Janis has a typing system that allows pipeline developers to be confident when they connect components together. In Janis, there are 6 fundamental types, all of which are inheritable. These fundamental types require an equivalent in all translation targets (eg: `janis.String` must be to a string CWL and WDL type).

- *`janis.File`*
- *`janis.String`*
- *`janis.Int`*
- *`janis.Float`*
- *`janis.Boolean`*
- `janis.Directory`

To provide greater context to your tools and workflows, you can create a custom type that inherits from all of these fundamental types.

Often you can just use the uninstantiated type for annotation. All types can be made optional by instantiating the type wth `(optional=True)`. By default types are required (non-optional) and inputs that have a default are made optional.

Patterns

The janis-patterns repository has an example of how python types relate to Janis types: https://github.com/PMCC-BioinformaticsCore/janis-patterns/blob/master/types/pythontypes.py

File

A file in Janis is an object, and not a path. By annotating your type as a file, the execution engine will make the referenced file available within your execution environment. You should simply consider that the File type is a reference to some arbitrary place on disk and not in a predictable location.

Most often you will not directly instantiate a File as you'll use some domain specific file type (such as `TarFile` or *Bam*').

class `janis.`**`File`**(*optional=False*, *extension=None*, *alternate_extensions: Set[str] = None*)

If you do not correctly annotate your type to be a *`janis.File`* (with appropriate index files), it may not be localised into your execution environment.

Secondary / Accessory files

Janis inherits the concept of secondary files from the Common Workflow Language. When creating a subclass of a File, you should include the

```
@staticmethod
def secondary_files() -> Optional[List[str]]:
    return None
```

And return a list of strings with the following syntax:

- If string begins with one or more caret ^ characters, for each caret, remove the last file extension from the path (the last period `.` and all following characters). If there are no file extensions, the path is unchanged.
- Append the remainder of the string to the end of the file path.

Note: It's important that if you're directly instantiating the File class, or subclassing and calling `super().__init__` that you MUST include the expected file extension, otherwise secondary files will be not correctly collected in WDL.

Typing system

The typing system allows for you to pass an inherited type to one of its super classes. This means that if your type *inherits* from `File`, you can safely pass your inherited type. When passing inherited file types with secondary files, Janis does it's best to take the subset of files from the receiving type. It's up to the developer's of new data types to ensure that all secondary files are included from each subtype.

String

A `String` is a type in Janis. When annotating a type, you can also use the inbuilt Python `str`, which will be turned into a required (non-optional) *`janis.String`*.

class `janis.`**`String`**(*optional=False*)

The *`janis.Filename`* class is a special type that inherits from string which allows for automatic filename generation.

Int

An `Int` is a type in Janis. When annotating a type, you can also use the inbuilt Python `int`, which will be turned into a required (non-optional) *`janis.Int`*.

class `janis.`**`Int`**(*optional=False*)

The range of an integer is the minimum of the specification you use, here are some values:

Language	Min	Max
CWL	`-2^32`	`2^32`
WDL	`-2^63`	`2^63`
Python	→	`sys.maxsize`
JSON	*	*
YAML	*	*

* - JSON and YAML max ranges may depend on your parser.

Float

A `Float` is a type in Janis. When annotating a type, you can also use the inbuilt Python `float`, which will be turned into a required (non-optional) *`janis.Float`*.

class `janis.`**`Float`**(*optional=False*)

The range of an integer is the minimum of the specification you use, here are some values:

Language	Min	Max
CWL	→	finite 32-bit IEEE-754
WDL	→	finite 64-bit IEEE-754
Python	→	`sys.float_info`
JSON	*	*
YAML	*	*

* - JSON and YAML max ranges may depend on your parser.

Boolean

A `Boolean` is a type in Janis. When annotating a type, you can also use the inbuilt Python `bool`, which will be turned into a required (non-optional) *`janis.Boolean`*.

class `janis.`**`Boolean`**(*optional=False*)

Filename

A `Filename` is an inherited type in Janis with the purpose to stand in for filename generation. By default it is always optional, and there are a number of ways you can customise it.

class `janis.`**`Filename`**(*prefix='generated'*, *suffix=None*, *extension: str = None*, *optional=None*)

- Prefix (defaults to "generated")
- Suffix
- Guid (defaults to evaluating `str(uuid.uuid1())` Overridable if you want a similar structure to your output files + some extension.
- Extension (this is very important if collecting output files), you must include the period (`.`).

Format: `$prefix-$guid-$suffix.bam`

The intention was for each run and scatter to have a different generated value for each instance of generated filename. However, at the moment technical difficulties has delayed this implementation.

Features:

4.16.6 Expressions

Note: This feature is only available in Janis-Core v0.9.x and above.

Expressions in Janis refer to a set of features that improve how you can manipulate and massage inputs into an appropriate structure.

In this set of changes, we consider two important types:

- `janis.Selector` A placeholder used for selecting other values (without modification).
- `janis.Operator` Analogous to a function call.

To remain abstract, Janis builds up a set of these operations, all of which have a CWL and WDL analogue. We'll talk more about how these are converted later.

Selectors

As initially described, a Selector can be seen as a _placeholder_ for referencing other values. Some common types of selectors:

- InputSelector - references the input by a string
  - MemorySelector - Special InputSelector for selecting memory in GBs
  - CpuSelector - Special InputSelector for getting number of CPUs that a task will request.
- InputNodeSelector - specifically references the input node, only available in a workflow
- StepOutputSelector - references the output of a step, only available in a workflow.
- WildcardSelector - Collect a number of files with a pattern. Currently the use of this pattern isn't well defined, but is passed without modification to CWL and WDL. Only available on a ToolOutput.

Selectors participate in the typesystem, and as such have a dynamic return type.

Operators

Operators are equivalent to the concept of functions. They take a set of inputs, and generate some output. Operators inherit from selectors, The inputs and outputs are typed, and hence operators and selectors particpate in the typing system.

Both are more flexible with the types that they provide, though some selectors (notable InputSelector) may be unable to give a definite type.

In terms of their implementation, it's the operator's responsibility to convert itself to CWL and WDL. There will be an example below.

In some cases (where possible), we've overridden the standard python operators to give the Janis analogue.

- `__neg__`
- `__and__` NB: this is the bitwise AND and not the logical `and`.
- `__rand__` NB: similar to __and__
- `__or__` NB: this is the bitwise OR and not the logical `or`.

- `__ror__`
- `__add__`
- `__radd__`
- `__sub__`
- `__rsub__`
- `__mul__`
- `__rmul__`
- `__truediv__`
- `__rtruediv__`
- `__eq__`
- `__ne__`
- `__gt__`
- `__ge__`
- `__lt__`
- `__le__`
- `__len__`
- `__getitem__`
- `to_string_formatter()`
- `as_str()`
- `as_bool()`
- `as_int()`
- `op_and()`
- `op_or()`
- `basename()`

List of operators

Note: See the List of operators guide for more information.

Example usage

An operator's usage should be as you'd expect, let's see an example use of the FileSizeOperator, highlighting:

- Import the operator from `janis_core.operators.standard`
- Applying the operation directly on the‘‘echo.inp‘‘ step input field.
- Converting the transformed input to a string with `.as_str()`

```
from janis_core import WorkflowBuilder, File
from janis_core.operators.standard import FileSizeOperator
from janis_unix.tools import Echo

w = WorkflowBuilder("sizetest")
w.input("fileInp", File)

w.step("print",
    Echo(inp=(FileSizeOperator(w.fileInp) * 1024).as_str())
)
w.output("out", source=w.print)
```

Before we go any further, let's look at the WDL and CWL translations:

WDL

We can see how the step input expression is converted directly inline.

```
version development

import "tools/echo.wdl" as E

workflow sizetest {
  input {
    File fileInp
  }
  call E.echo as print {
    input:
      inp=((1024 * size(fileInp, "MB")))
  }
  output {
    File out = print.out
  }
}
```

CWL

The CWL translation is a little bit trickier due to the way scope of `valueFrom` expressions. We can see that our variable is passed into the scope (prefixed by an underscore), and then the valueFrom contains the expression that we produced - this is the value that the `print` step will see.

```
 #!/usr/bin/env cwl-runner
 class: Workflow
 cwlVersion: v1.0

 requirements:
   InlineJavascriptRequirement: {}
   StepInputExpressionRequirement: {}

 inputs:
inp:
     type: File

 outputs:
```

(continues on next page)

(continued from previous page)

```
out:
     type: File
     outputSource: print/out

steps:
  print:
    label: Echo
    in:
      _print_inp_fileInp:
        source: fileInp
      inp:
        valueFrom: $(String((1024 * (inputs._print_inp_fileInp.size / 1048576))))
    run: tools/echo.cwl
    out:
    - out
id: sizetest
```

Implementation notes

Let's first look at the implementation of the `janis_core.operators.standard.FileSizeOperator`:

For example, we could consider the implementation of the `FileSizeOperator`:

```
class FileSizeOperator(Operator):
    """
    Returned in MB: Note that this does NOT include the reference files (yet)
    """

    def argtypes(self):
        return [File()]

    def returntype(self):
        return Float

    def __str__(self):
        f = self.args[0]
        return f"file_size({f})"

    def to_wdl(self, unwrap_operator, *args):
        f = unwrap_operator(self.args[0])
        return f'size({f}, "MB")'

    def to_cwl(self, unwrap_operator, *args):
        f = unwrap_operator(self.args[0])
        return f"({f}.size / 1048576)"
```

- The argtypes returns an array of the types of the arguments that we expect. FileSizeOperator only expects one arg of a File type
- The ReturnType is a singular field which depicts the return type of the function.
- `to_wdl` is the function that builds our WDL function, it calls `unwrap_operator` on the argument (to ensure that any tokens are unwrapped), and then builds the command line using string interpolation.
- `to_cwl` operates exactly the same, except we use ES5 javascript to build our operation. The `/ 1048576` is to ensure that the value we receive in Bytes is converted to Megabytes (MB).

PROPOSED

In order to keep the current spec scoped, there is additional functionality that is planned:

- Using operators to build up the CPU, Memory and future resource values
- Custom python operations

4.16.7 Selectors and StringFormatting

Methods to reference or transform inputs

Overview

A `janis.Selector` is an abstract class that allows a workflow author to reference or transform a value. There are 3 main forms of selectors:

- *`janis.InputSelector`* - Used for selecting inputs
- *`janis.WildcardSelector`* - Used for collecting outputs
- *`janis.StringFormatter`* - Constructing or transforming strings

InputSelector

Declaration

class janis.**InputSelector**(*input_to_select*, *remove_file_extension=None*, *type_hint=<class ’janis_core.types.common_data_types.File’>*, ***kwargs*)

Overview

An `InputSelector` is used to get the value of an input at runtime. This has a number of use cases:

- Collecting outputs through the `glob=` field.
- Constructing a *`janis.StringFormatter`* from existing inputs

These inputs are checked for validity at runtime. In CWL, defaults are propagated by definition in the language spec, in WDL this propagation is handled by janis.

Example

The following code will use the value of the `outputFilename` to find the corresponding output to `bamOutput`. Additionally as the data_type `BamBai` has a secondary file, in WDL janis will automatically construct a second output corresponding to the `.bai` secondary (called `bamOutput_bai` and modify the contents of the `InputSelector` to correctly glob this index file.

```python
def outputs(self):
    return [
        ToolOutput("bamOutput", BamBai, glob=InputSelector("outputFilename"))
    ]
```

WildcardSelector

Declaration

class `janis.`**`WildcardSelector`**(*wildcard*, *select_first=False*)

Overview

A wildcard selector is used to glob a collection of files for an output where an `InputSelector` is not appropriate. Note that if an `*` (asterisk | star) bind is used to collect files, the output type should be an array.

Example

The following example will demonstrate how to get all the files ending on `.csv` within the execution directory.

```
def outputs(self):
    return [
        ToolOutput("metrics", Array(Csv), glob=WildcardSelector("*.csv"))
    ]
```

StringFormatting

Declaration

class `janis.`**`StringFormatter`**(*format: str*, ***kwargs*)

Overview

A `StringFormatter` is used to allow inputs or other values to be inserted at runtime into a string template. A `StringFormatter` can be concatenated with Python strings, another `StringFormatter` or an `InputSelector`.

The string `"{placeholdername}"` can be used within a string format, where `placeholdername` is a `kwarg` passed to the StringFormatter with the intended selector or value.

The placeholder names must be valid Python variable names (as they're passed as kwargs). See the String formatter tests for more examples.

Example

- Initialisation

  `StringFormatter("Hello, {name}", name=InputSelector("username"))`

- Concatenation

  `"Hello, " + InputSelector("username")`   `InputSelector("greeting") + StringFormatter(", {name}", name=InputSelector("username"))`

4.16.8 Scattering

Improving workflow performance with embarrassingly parallel tasks

Janis support scattering by field when constructing a *janis.Workflow.step()* through the `scatter=Union[str, janis.ScatterDescription ]` parameter.

class janis.**ScatterDescription**(*fields: List[str], method: janis_core.utils.scatter.ScatterMethod = None, labels: Union[janis_core.operators.selectors.Selector, List[str]] = None*)

Class for keeping track of scatter information

__init__(*fields: List[str], method: janis_core.utils.scatter.ScatterMethod = None, labels: Union[janis_core.operators.selectors.Selector, List[str]] = None*)

Parameters

- **fields** – The fields of the the tool that should be scattered on.
- **method** (*ScatterMethod*) – The method that should be used to scatter the two arrays
- **labels** – (JANIS ONLY) -

janis.**ScatterMethods**

alias of `janis_core.utils.scatter.ScatterMethod`

Simple scatter

To simply scatter by a single field, you can simple provide the `scatter="fieldname"` parameter to the `janis.Workflow.step()` method.

For example, let's presume you have the tool `MyTool` which accepts a single string input on the `myToolInput` field.

```
w = Workflow("mywf")
w.input("arrayInp", Array(String))
w.step("stp", MyTool(inp1=w.arrayInp), scatter="inp1")
# equivalent to
w.step("stp", MyTool(inp1=w.arrayInp), scatter=ScatterDescription(fields=["inp1"]))
```

Scattering by more than one field

Janis supports scattering by multiple fields by the `dot` and `scatter` methods, you will need to use a *janis.ScatterDescription* and *janis.ScatterMethods*:

Example:

```
from janis import ScatterDescription, ScatterMethods
# OR
from janis_core import ScatterDescription, ScatterMethods

w = Workflow("mywf")
w.input("arrayInp1", Array(String))
w.input("arrayInp2", Array(String))
w.step(
  "stp",
  MyTool(inp1=w.arrayInp1, inp2=w.arrayInp2),
  scatter=ScatterDescription(fields=["inp1", "inp2"], method=ScatterMethods.dot)
)
```

4.17 Janis Prepare

- # * = - .

`prepare` is functionality of Janis to improve the process of running pipelines. Specifically, Janis prepare will perform a few key actions:

- Downloads reference datasets
- Performs simple transforms data into the correct format (eg: VCF -> gVCF)
- **Checks quality of some inputs:**
  - for example, contigs in a bed file match what's in a declared reference

Note: Prepare only works with Janis pipelines

4.17.1 Quickstart

The `janis prepare` command line is almost exactly the same as the `janis run`. You should supply it with inputs (either through an inputs yaml or on the command line) and any other configuration options. You must supply an output directory (or declare an `output_dir` in your janis config), the job file and a run script is written to this directory. The job file is also always written to stdout.

For example:

```
# Write inputs file
cat <<EOT >> inputs.yaml
sampleName: NA12878
fastqs:
- - /<fastqdata>/WGS_30X_R1.fastq.gz
  - /<fastqdata>//WGS_30X_R2.fastq.gz
EOT

# run janis prepare
janis prepare \
    -o "$HOME/janis/WGSGermlineGATK-run/" \
    --inputs inputs.yaml \
    --source-hint hg38 \
    WGSGermlineGATK
```

This will: - Write an inputs file to disk, - download all the hg38 reference files - transform any data types that might need to be transformed, eg:

- `gridss blacklist` requires a bed, but the source hint gives a gzipped bed (`ENCFF001TDO.bed.gz`)
- `snps_dbsnp` wants a compressed and tabix indexed VCF, but the source hint gives a regular VCF.
- `reference` build the appropriate indexes for the hg38 assembly (as these aren't downloaded by default)
- **Perform some sanity checks on the data you've provided, eg:**
  - the contigs in the gridss_blacklist will be checked against those found in the assembly's reference.
- Write a job file and run script into the output directory.

4.17.2 Downloading reference datasets

A `source` is declared on an input through the input documentation, there are two methods for doing this:

1. Single file - this file is always used regardless of the inputs
2. A dictionary of *source hints*. This might be useful for specifying a reference type, eg hg38 or hg19. However this does involve the pipeline author finding public datasets for each *hint*.

Example:

```
import janis_core as j

myworkflow = j.WorkflowBuilder("wf")

# Input that can be used for ALL workflows
myworkflow.input(
    "inp",
    File,
    doc=j.InputDocumentation(
        source="https://janis.readthedocs.io/en/latest/references/prepare.html"
    )
)

# Input that is only localised if the hint is specified
mworkflow.input(
    "gridss_blacklist",
    File,
    doc=j.InputDocumentation(
        source={
            "hg19": "https://www.encodeproject.org/files/ENCFF001TDO/@@download/
↪ENCFF001TDO.bed.gz",
            "hg38": "https://www.encodeproject.org/files/ENCFF356LFX/@@download/
↪ENCFF356LFX.bed.gz",
        }
    ),
)
```

Secondary files and localising sources

By default, secondary files are downloaded with the primary file. The pipeline author can optionally request that secondary files be not downloaded with `skip_sourcing_secondary_files=True`. In particular, this is important for the exemplar WGS pipelines, because the BWA indexes (`".amb", ".ann", ".bwt", ".pac", ".sa"`) were generated with a different version of BWA.

Although the Janis Transformation step of the janis prepare will perform the re-index, this could take a few hours. If you can speed up this step, please *raise an issue <https://github.com/PMCC-BioinformaticsCore/janis-bioinformatics/issues/new>* or open a Pull Request!!

Types of reference paths

Janis can localise remote paths through its *FileScheme* mechanic. Currently, Janis supports the following fileschemes:

- Local (useful for keeping local references to a pipeline definition when sharing a pipeline internally)
- HTTP `http://  OR http://` prefix (using a GET request)
- GCS `gs://` prefix (public buckets only)

Requires more work: - S3: Implementation required

Input Documentation Source

An `InputDocumentation` class can be used to document the different options about an input. See the *:class:janis.InputDocumentation* initialiser below. You can supply it directly to the doc field of an input, but you can also provide a dictionary which will get converted into an InputDocumentation, for example:

```
w = j.WorkflowBuilder("wf")

# using the InputDocumentation class
w.input("inp1", str, doc=j.InputDocumentation(doc="This is inp1", quality=j.
↪InputQualityType.user))

# Use a dictionary
w.input("inp2", str, doc={"doc": "This is inp2", "quality": "user"})
```

class janis.**InputDocumentation**(*doc: Optional[str], quality: Union[janis_core.tool.documentation.InputQualityType, str] = <InputQualityType.user: 'user'>, example: Union[str, List[str], None] = None, source: Union[str, List[str], Dict[str, Union[str, List[str]]], None] = None, skip_sourcing_secondary_files=False*)

__init__(*doc: Optional[str], quality: Union[janis_core.tool.documentation.InputQualityType, str] = <InputQualityType.user: 'user'>, example: Union[str, List[str], None] = None, source: Union[str, List[str], Dict[str, Union[str, List[str]]], None] = None, skip_sourcing_secondary_files=False*)

Extended documentation for inputs

Parameters

- **doc** (*str*) – Documentation string
- **quality** (*InputQualityType | "user" | "static" | "configuration"*) – quality of input, whether the inputs are best classified by user (data), static (references), configuration (like constants, but tweakable)
- **example** (*str | List[str]*) – An example of the filename, displayed in the generated example input.yaml
- **source** (*str | List[str] | Dict[str, str | List[str]]*) – A URI of this input, that Janis could localise if it's not provided. For example, you might want to specify a gs://<path>
- **skip_sourcing_secondary_files** (*bool*) – Skip localising the secondary files from the source. You might want to do this if the secondary files depend on the version of the tool (eg: BWA)

4.17.3 Transformations

A strong benefit of the rich data types that Janis provides, means that we can do perform basic transformations to prepare your data for the pipeline. Simply, we've taught Janis how to perform some basic transformations (using tools in Janis!), and then Janis can determine how to convert your data, for example:

If you provide a VCF, and the pipeline requires a gzipped and tabix'd VCF, we construct a prepare stage that performs this for you:

```
VCF -> VcfGZ -> VcfGzTabix
```

More information

- The Conversion Problem (Janis guide)
- Bioinformatics transformations github:janis-bioinformatics/transformations/__init__.py

Note: These Janis transformations are performed AFTER the reference localisation, so janis can transform your downloaded file to the correct format if possible.

4.17.4 Quality Checks

These are a number of fairly custom checks to catch some frequent errors that we've seen:

- Input checker: makes sure your inputs can be found (only works for local paths)
- Contig checker: If you define a single input of `FASTA` type and any number of inputs with `BED` type, Janis will check that the contigs you declare in the `BED`s are found in the `.fasta.fai` index. This check only returns warnings, and will NOT stop you from running a workflow.

Janis performs some of these checks on every run, some are only performed on the *janis prepare*.

4.17.5 Developer notes

The Janis prepare steps are all implemented using `PipelineModifiers`:

- `FileFinderLocatorModifier`
- `InputFileQualifierModifier`
- `InputTransformerModifier`
- `InputChecker`
- `ContigChecker`

If entering through the *prepare* cli method, the `run_prepare_processing` flag is set which initialises a custom set of pipeline modifiers to execute while preparing the job file.

Janis Transformations

Janis transformations are fairly simple, they're defined in the relevant tool registry (eg: janis-bioinformatics), and exposed through the `janis.datatype_transformations` entrypoint. These JanisTransformations are added to a graph, and then we just perform a breadth first search on this graph looking for the shortest number of steps to connect two data types. Transformations are directional, and no logic is performed to evaluate the *weight* or *effort* of a step.

The Conversion Problem (Janis guide) describes JanisTransformations in a blog sort of style.

Job File

Although not strictly related to Janis Prepare, it was an important change that was made for janis-prepare to work correctly. Functionally, a `PreparedJob` describes everything needed to run a workflow (except the workflow).

It's an amalgamation of different sections of the janis configuration, adjusted inputs and runtime configurations. It's serializable, which means it can automatically be written to disk and parsed back in.

A `run.sh` script can be generated, which just calls the job file with the workflow reference, something like:

```
# This script was automatically generated by Janis on 2021-01-13 11:14:40.121673.

janis run \
    -j /Users/franklinmichael/janis/hello/20210113_111440/janis/job.yaml \
    hello
```

4.18 Secondary / Accessory Files

In some domains (looking specifically at Bioinformatics here), a single file isn't enough to contain all the information. Janis borrows the concept of secondary files from the CWL specification, and in fact we use the same pattern for grouping these files.

Often a *secondary* or *accessory* file is used to provide additional information, potentially a quick access index. These files are attached to the original file by a specific file pattern. We'll talk about that more soon.

For this reason, Janis allows data_types that inherit from a `File` to specify a `secondary_file` list of files to be bundled with.

4.18.1 Secondary file pattern

As earlier mentioned, we follow the Common Workflow Language secondary file pattern:

1. If string begins with one or more caret ^ characters, for each caret, remove the last file extension from the path (the last period `.` and all following characters). If there are no file extensions, the path is unchanged.
2. Append the remainder of the string to the end of the file path.

Examples

- **IndexedBam**
  - Pattern: `.bai`
  - **Files:**
    - Base: `myfile.bam`
    - `myfile.bam.bai`
- **FastaWithIndexes:**
  - Pattern: `.amb`, `.ann`, `.bwt`, `.pac`, `.sa`, `.fai`, `^.dict`
  - **Files:**
    - Base: `reference.fasta`
    - `reference.fasta.amb`
    - `reference.fasta.ann`
    - `reference.fasta.bwt`
    - `reference.fasta.pac`

- `reference.fasta.sa,`
- `reference.fasta.fai`
- `reference.dict`

Proposed

Implement optional secondary files as per CWL v1.1.

4.18.2 Implementation note

CWL

As we mimic the CWL secondary file pattern, we don't need to do any extra work except by providing this pattern to a:

- CommandInputParameter
- CommandOutputParameter
- InputParameter
- WorkflowOutputParameter

If you use the `secondaries_present_as` on a *`janis.ToolInput`* or *`janis.ToolOutput`*, a CWL expression is generated to rename the secondary file expression. More information can be found about this in common-workflow-language/cwltool#1232.

Issues

CWLTool has an issue when attempting to scatter using multiple fields (using the `dotproduct` or `*_crossproduct` methods), more information can be found on common-workflow-language/cwltool#1208.

WDL

The translation for WDL to implement secondary files was one of the most challenging aspects of the translation. Notably, WDL has no concept of secondary files. There are a few things we had to consider:

- Every file needs to be individually localised.
- A data type with secondary files can be used in an array of inputs
- Secondary files may need to be globbed if used as an Output data type
- An array of files with secondaries can be scattered on (including scattered by multiple fields)
- Janis should fill the input job with these secondary files (with the correct extension)

Implementation

Let's just break this down into different sections

Case 1: Simple index

The following `workflow.input("my_bam", BamBai)`, definition when connected to a tool might look like the following

```
workflow WGSGermlineGATK {
  input {
    File my_bam
    File my_bam_bai

  }
  call my_tool {
    input:
      bam=my_bam
      bam_bai=my_bam_bai
  }
  output {
    File out_bam = my_tool.out
    File out_bam_bai = my_tool.out_bai
  }
}
```

Note the extra annotations and mappings fot the `bai` type.

Case 2: Array of inputs with simple scatter

This is modification of the first example, nb: this isn't full functional workflow code:

```
workflow.input("my_bams", Array(BamBai))

workflow.step(
    "my_step",
    MyTool(bam=workflow.my_bams),
    scatter="bam"
)
```

Might result in the following workflow:

```
workflow WGSGermlineGATK {
  input {
    Array[File] my_bams
    Array[File] my_bams_bai

  }
  scatter (Q in zip(my_bams, my_bams_bai)) {
    call my_tool as my_step {
      input:
        bam=Q.left
        bam_bai=Q.right
    }
  }

  output {
    Array[File] out_bams = my_tool_that_accepts_array.out
    Array[File] out_bams_bai = my_tool_that_accepts_array.out_bai
```

(continues on next page)

(continued from previous page)

```
  }
}
```

Case 3: Multiple array inputs, scattering by multiple fields

Consider the following workflow:

```
workflow.input("my_bams", Array(BamBai))
workflow.input("my_references", Array(FastaBwa))

workflow.step(
    "my_step",
    ToolTypeThatAcceptsMultipleBioinfTypes(
        bam=workflow.my_bams, reference=workflow.my_references
    ),
    scatter=["bam", "reference"],
)

workflow.output("out_bam", source=workflow.my_step.out_bam)
workflow.output("out_reference", source=workflow.my_step.out_reference)
```

This gets complicated quickly:

```
workflow scattered_bioinf_complex {
  input {
    Array[File] my_bams
    Array[File] my_bams_bai
    Array[File] my_references
    Array[File] my_references_amb
    Array[File] my_references_ann
    Array[File] my_references_bwt
    Array[File] my_references_pac
    Array[File] my_references_sa
  }
  scatter (Q in zip(transpose([my_bams, my_bams_bai]), transpose([my_references, my_
↪references_amb, my_references_ann, my_references_bwt, my_references_pac, my_
↪references_sa]))) {
     call MyTool as my_step {
      input:
        bam=Q.left[0],
        bam_bai=Q.left[1],
        reference=Q.right[0],
        reference_amb=Q.right[1],
        reference_ann=Q.right[2],
        reference_bwt=Q.right[3],
        reference_pac=Q.right[4],
        reference_sa=Q.right[5]
    }
  }
  output {
    Array[File] out_bam = my_step.out_bam
    Array[File] out_reference = my_step.out_reference
  }
}
```

Known limitations

- **There is no namespace collision:**
  - Two files with similar prefixes but differences in punctuation will clash
  - A second input that is suffixed with the secondary's extension will clash: eg: mybam_bai will clash with mybam with a secondary of .bai.
- **Globbing a secondary file might not be possible when the original file extension is unknown. There are 2 considerations fo**
  - Subclasses of File should caller super() with the expected extension
  - Globbing based on a generated Filename (through InputSelector), will consider the extension property.

Relevant WDL issues:

- broadinstitute/cromwell#2269 (Secondary index files and directories in WDL)
- openwdl/wdl#289 (File Bundles and Secondary / Accessory Files)
- GATK Forums#9299 (Secondary index files and directories in WDL)

4.19 Unit Test Framework

Note: Available in v0.11.0 and later

4.19.1 Overview

You can write test cases for your tool by defining the `tests()` function in your Tool class. Test cases defined by this function will be picked up by `janisdk run-test` command.

This unit test framework provides several predefined preprocessors to transform Janis execution output data in the format that can be tested. For example, there is a preprocessor to read the md5 checksum of an output file. In addition to the predefined preprocessors, the frameworks also allows users to define and pass their own preprocessors.

Define test cases

You can define multiple test cases per tool. For each test case, you can declare multiple expected outputs. Mostly, you really only need to define more test cases if they require different input data.

The classes we use here *janis_core.tool.test_classes.TTestCase*, *janis_core.tool.test_classes.TTestExpectedOutput* and *janis_core.tool.test_classes.TTestPreprocessor* are declared at the bottom of this document.

```
class BwaAligner(BioinformaticsWorkflow):
    def id(self):
        return "BwaAligner"

    ...

    def tests(self):
        return [
```

(continues on next page)

(continued from previous page)

```
            TTestCase(
                name="basic",
                input={
                    "bam": "https://some-public-container/directory/small.bam"
                },
                output=[
                    TTestExpectedOutput(
                        tag="out",
                        preprocessor=TTestPreprocessor.FileMd5,
                        operator=operator.eq,
                        expected_value="dc58fe92a9bb0c897c85804758dfadbf",
                    ),
                    TTestExpectedOutput(
                        tag="out",
                        preprocessor=TTestPreprocessor.FileContent,
                        operator=operator.contains,
                        expected_value="19384 + 0 in total (QC-passed reads + QC-
↪failed reads)",
                    ),
                    TTestExpectedOutput(
                        tag="out",
                        preprocessor=TTestPreprocessor.LineCount,
                        operator=operator.eq,
                        expected_value=13,
                    ),
                ],
            )
        ]
```

Run the tests

```
# Run a specific test case
janisdk run-test --test-case [TEST CASE NAME] [TOOL ID]

# Run ALL test cases of one tool
janisdk run-test [TOOL ID]
```

To run the example test case shown above:

```
# Run a specific test case
janisdk run-test --test-case=basic BwaAligner

# Run all test cases
janisdk run-test BwaAligner
```

4.19.2 Test Files

There are two different ways to store your test files (input and expected output files):

Remote HTTP files:

you can use a publicly accessible http link `https://some-public-container/directory/small.bam`.

- Input files will be downloaded to a cache folder in `~/.janis/remote_file_cache` folder. This is the same directory where files will be cached when you run `janis run`.
- Expected output files however will be cached in the test directory `[WORKING DIRECTORY WHERE janisdk run-test is run]/tests_output/cached_test_files/`.

If the same url is found in the cache directory, we will not re-download the files unless the `Last-Modified` http header has changed. If you want to force the files to be re-downloaded, you will need to remove the files from the cache directories.

Local test files:

you can store your files in local directory named `test_data`. There are a few different examples of where you can place this directory. Example from `janis-bioinformatics` project:

- A `test_data` folder that contain files to be shared by multiple tools can be located at `janis_bioinformatics/tools/test_data`. To access files in this directory, you can call `os.path.join(BioinformaticsTool.test_data_path(), "small.bam")`.
- A `test_data` folder that contain files to be used by `flagstat` can be located at `janis_bioinformatics/tools/samtools/flagstat/test_data`. To access files in this directory from within the `SamToolsFlagstatBase` class, you can call `os.path.join(self.test_data_path(), "small.bam")`.

4.19.3 Preprocessors and Comparison Operators

`TTestExpectedOutput.preprocessor` is used to reformat the Tool output. `TTestExpectedOutput.operator` is used to compare output value with the expected output.

Predefined Preprocessors

- **Value**: No preprocessing, value as output by Janis e.g. an integer, string, or a file path for a File type output.
- **FileDiff**: The differences between two files as output by `difflib.unified_diff`. This can only be applied on File type output. If this preprocessor is used, `TTestExpectedOutput.file_diff_source` must be provided. `file_diff_source` must contain the file path to compare the output file with.
- **LinesDiff** Number of different lines between two files as a tuple `(additions, deletions)`, as diff'd by the `FileDiff` preprocessor. If this preprocessor is used, `TTestExpectedOutput.file_diff_source` must be provided. `file_diff_source` must contain the file path to compare the output file with.
- **FileContent**: Extract the file content. This can only be applied to File type output.
- **FileExists**: Check if a file exists. It returns a True/False value. This can only be applied to File type output.
- **FileSize**: File size is bytes. This can only be applied on File type output.
- **FileMd5**: Md5 checksum of a file. This can only be applied to File type output.
- **LineCount**: Count the number of lines in a string or in a file.
- **ListSize**: Count the number of items in a list. This can only be applied to Array type output.

Custom preprocessor example:

In this example below, we are testing a tool that has an output field named `out`. The output value of this field is a file path that points to the location of a BAM file. We want to test the flagstat value of this BAM file. Here, we define your custom preprocessor function that takes a file path as input and returns a string that contains the flagstat value of a BAM file.

`TTestExpectedOutput.expected_file` simply points to a file that contains the expected output value. You can also replace this with `TTestExpectedOutput.expected_value="19384 + 0 in total (QC-passed reads + QC-failed reads)\n ..."`

```
TTestExpectedOutput(
    tag="out",
    preprocessor=Bam.flagstat,
    operator=operator.eq,
    expected_file="https://some-public-container/directory/flagstat.txt"
)
```

```
class Bam(File):
    ...

    @classmethod
    def flagstat(cls, file_path: str):
        command = ["samtools", "flagstat", file_path]
        result = subprocess.run(
            command,
            stdout=subprocess.PIPE,
            stderr=subprocess.PIPE,
            universal_newlines=True,
        )

        if result.stderr:
            raise Exception(result.stderr)

        return result.stdout
```

Custom operator example:

In this example, we also want to test the flagstat value of BAM file returned by the `out` output field.

Here, instead of writing a custom preprocessors, we write a custom operator that takes two file path and compare the flagstat output of this two files.

```
TTestExpectedOutput(
    tag="out",
    preprocessor=TTestPreprocessor.Value,
    operator=Bam.equal,
    expected_value="https://some-public-container/directory/small.bam"
)
```

```
class Bam(File):
    ...

    @classmethod
    def equal(cls, file_path_1: str, file_path_2: str):
```

(continues on next page)

(continued from previous page)

```
        flagstat1 = cls.flagstat(file_path_1)
        flagstat2 = cls.flagstat(file_path_2)

        return flagstat1 == flagstat2
```

4.19.4 Declaration

class janis_core.tool.test_classes.**TTestCase**(*name: str, input: Dict[str, Any], output: List[janis_core.tool.test_classes.TTestExpectedOutput]*)

A test case requires a workflow or tool to be run once (per engine). But, we can have multiple output to apply different test logic.

class janis_core.tool.test_classes.**TTestExpectedOutput**(*tag: str, preprocessor: Union[janis_core.tool.test_classes.TTestPreprocesso Callable[[Any], Any]], operator: Callable[[Any, Any], bool], expected_value: Optional[Any] = None, expected_file: Optional[str] = None, file_diff_source: Optional[str] = None, array_index: Optional[int] = None, suffix_secondary_file: Optional[str] = None, preprocessor_params: Optional[Dict[KT, VT]] = {}*)

Describe the logic on how to test the expected output of a test case. A test case can have multiple instances of this class to test different output or different logic of the same output

class janis_core.tool.test_classes.**TTestPreprocessor**

An Enum class for a list of possible pre-processing we can do to output values

4.20 Configuring Janis Assistant

> **Warning:** Configuring Janis had backwards incompatible changes in v0.12.0 with regards to specific keys.

When we talk about configuring Janis, we're really talking about how to configure the Janis assistant and hence Cromwell / CWLTool to interact with batch systems (eg: Slurm / PBS Torque) and container environments (Docker / Singularity).

Janis has built in templates for the following compute environments:

- **Local (Docker or Singularity)**
    - The only environment compatible with CWLTool.
- Slurm (Singularity only)
- PBS / Torque (Singularity only)

In an extension of Janis (janis-templates), we've produced a number of location specific templates, usually with sensible defaults.

See this list for a full list of templates: https://janis.readthedocs.io/en/latest/templates/index.html.

4.20.1 Syntax

The config should be in YAML, and can contain nested dictionaries.

Note: In this guide we might use dots (.) to refer to a nested key. For example, notifications.email refers to the following structure:

```
notifications:
  email: <value>
```

4.20.2 Location of config

By default,

- Your configuration directory is placed at ~/.janis/.
- Your configuration path is a file called janis.conf in this directory, eg: ~/.janis/janis.conf

Both janis run / translate allow you to provide a location to a config using the -c / --config parameter.

In addition, you can also configure the following environment variables:

- JANIS_CONFIGPATH - simply the path to your config file
- JANIS_CONFIGDIR - The configuration path is determined by *$JANIS_CONFIGDIR/janis.conf*

4.20.3 Initialising a template

`` `bash janis init <template> [...options] ` ``

By default this will place our config at *~/.janis/janis.conf*. If there's already a config there, it will NOT override it.

- -o / --output: output path to write to, default (*~/.janis/janis.conf*).
- -f / --force: Overwrite the config if one exists at the output path.
- --stdout: Write the output to stdout.

4.20.4 Environment variables

The following environment variables allow you to customise the behaviour of Janis, without writing a configuration file. This is a convenient way to automatically configure Janis through an HPCs module system.

All the environment variables start with JANIS_, and is the value of enums declared below (and not the key name), for example:

```
export JANIS_CROMWELLJAR='/path/to/cromwell.jar'
```

class janis_assistant.management.envvariables.**EnvVariables**

An enumeration.

config_dir = 'JANIS_CONFIGDIR'
(Default: ~/.janis) Directory of default Janis settings

config_path = 'JANIS_CONFIGPATH'
(Default: `$JANIS_CONFIGDIR/janis.conf`) Default configuration file for Janis

cromwelljar = 'JANIS_CROMWELLJAR'
Override the Cromwell JAR that Janis uses

default_template = 'JANIS_DEFAULTTEMPLATE'
Default template to use, NB this template should have NO required arguments.

exec_dir = 'JANIS_EXCECUTIONDIR'
Use this directory for intermediate files

output_dir = 'JANIS_OUTPUTDIR'
Use this directory as a BASE to generate a new output directory for each Janis run

recipe_directory = 'JANIS_RECIPEDIRECTORY'
Directories for which each file (ending in .yaml | .yml) is a key of input values. See the RECIPES section for more information.

recipe_paths = 'JANIS_RECIPEPATHS'
List of YAML recipe files (comma separated) for Janis to consume, See the RECIPES section for more information.

search_path = 'JANIS_SEARCHPATH'
Additional search paths (comma separated) to lookup Janis workflows in

4.20.5 Configuration keys

We use a class definition to automatically document which keys you can provide to build a Janis configuration. These keys exactly match the keys you should provide in your YAML dictionary. The type is either a Python literal, or a dictionary.

For example, the class definition below corresponds to the following (partial) YAML configuration:

```yaml
# EXAMPLE CONFIGURATION ONLY
engine: cromwell
run_in_background: false
call_caching_enabled: true
cromwell:
    jar: /Users/franklinmichael/broad/cromwell-53.1.jar
    call_caching_method: fingerprint
```

class janis_assistant.management.configuration.**JanisConfiguration**(*type*)

__init__(*output_dir: str = None, execution_dir: str = None, call_caching_enabled: bool = True, engine: str = 'cromwell', cromwell: Union[janis_assistant.management.configuration.JanisConfigurationCromwell, dict] = None, template: Union[janis_assistant.management.configuration.JanisConfigurationTemplate, dict] = None, recipes: Union[janis_assistant.management.configuration.JanisConfigurationRecipes, dict] = None, notifications: Union[janis_assistant.management.configuration.JanisConfigurationNotifications, dict] = None, environment: Union[janis_assistant.management.configuration.JanisConfigurationEnvironment, dict] = None, run_in_background: bool = None, digest_cache_location: str = None, container: Union[str, janis_assistant.containers.base.Container] = None, search_paths: List[str] = None*)

Parameters

- **engine** (*"cromwell" | "cwltool"*) – Default engine to use
- **template** (JanisConfigurationTemplate) – Specify options for a Janis template for configuring an execution environment
- **cromwell** (JanisConfigurationCromwell) – A dictionary for how to configure Cromwell for Janis
- **recipes** (JanisConfigurationRecipes) – Configure recipes in Janis
- **notifications** (JanisConfigurationNotifications) – Configure Janis notifications
- **environment** (JanisConfigurationEnvironment) – Additional ways to configure the execution environment for Janis
- **output_dir** – A directory that Janis will use to generate a new output directory for each janis-run
- **execution_dir** – Move all execution to a static directory outside the regular output directory.
- **call_caching_enabled** – (default: true) call-caching is enabled for subsequent runs, on the SAME output directory
- **run_in_background** (*bool*) – By default, run workflows as a background process. In a SLURM environment, this might submit Janis as a SLURM job.
- **digest_cache_location** (*str*) – A cache of docker tags to its digest that Janis uses replaces your docker tag with it's digest.
- **container** (*"docker" | "singularity"*) – Container technology to use, important for checking if container environment is available and running mysql instance.
- **search_paths** (*List[str]*) – A list of paths to check when looking for python files and input files

Template

Janis templates are a convenient way to handle configuring Janis, Cromwell and CWLTool for special environments. A number of templates are prebuilt into Janis, such as `slurm_singularity`, `slurm_pbs`, and number of additional templates for specific HPCs (like Peter Mac, Spartan at UoM) are available, and documented in the:

- List of templates page.

You could use a template like the following:

```
# rest of janis configuration
template:
  id: slurm_singularity
  # arguments for template 'slurm_singularity', like 'container_dir'.
  container_dir: /shared/path/to/containerdir/
```

class janis_assistant.management.configuration.**JanisConfigurationTemplate**(*type*)

__init__(*id: str = None, **d*)

Parameters **id** (*The identifier of the template*) –

Cromwell

Sometimes Cromwell can be hard to configure from a simple template, so we've exposed some extra common options. Feel free to [raise an issue](https://github.com/PMCC-BioinformaticsCore/janis-assistant/issues/new) if you have questions or ideas.

class `janis_assistant.management.configuration.`**`JanisConfigurationCromwell`**(*type*)

`__init__`(*jar: str = None*, *config_path: str = None*, *url: str = None*, *memory_mb: int = None*, *call_caching_method: str = 'fingerprint'*, *timeout: int = 10*, *polling_interval=None*, *db_type: janis_assistant.data.enums.dbtype.DatabaseTypeToUse = <DatabaseTypeToUse.filebased: 'filebased'>*, *mysql_credentials: Union[dict, janis_assistant.management.configuration.MySqlInstanceConfig] = None*, *additional_config_lines: str = None*)

Parameters

- **`url`** (*`str`*) – Use an existing Cromwell instance with this URL (with port). Use the BASE url, do NOT include http.
- **`jar`** – Specific Cromwell JAR to use (prioritised over `$JANIS_CROMWELLJAR`)
- **`config_path`** – Use a supplied Config path when running a Cromwell instance. Also see `additional_config_lines` for including specific cromwell options.
- **`memory_mb`** – Amount of memory to give Cromwell instance through `java -xmx <max-memory>M -jar <jar>`
- **`call_caching_method`** – (Default: "fingerprint") Cromwell caching strategy to use, see Call cache strategy options for local filesystem for more information.
- **`timeout`** – Suspend a Janis workflow if unable to contact cromwell for <timeout> MINTUES.
- **`polling_interval`** – How often to poll Cromwell, by default this starts at 5 seconds, and gradually falls to 60 seconds over 30 minutes. For more information, see the `janis_assistant.Cromwell.get_poll_interval` method
- **`db_type`** (*`"none" | "existing" | "managed" | "filebased" | "from_script"`*) – (Default: filebased) DB type to use for Janis. "none" -> no database; "existing" -> use mysql credentials from `cromwell.mysql_credentials`; "managed" -> Janis will start and manage a containerised MySQL instance; "filebased": Use the HSQLDB filebased db through Cromwell for SMALL workflows only (NB: this can produce large files, and timeout for large workflows); "from_script": Call the script `$JANIS_DBCREDENTIALSGENERATOR` for credentials. See get_config_from_script for more information.
- **`mysql_credentials`** (`MySqlInstanceConfig`) – A dictionary of MySQL credentials
- **`additional_config_lines`** (*`str`*) – A string to add to the bottom of a generated Cromwell configuration. This is NOT used for an existing cromwell instance, or a config is supplied.

class `janis_assistant.management.configuration.`**`MySqlInstanceConfig`**(*type*)

`__init__`(*url*, *username*, *password*, *dbname='cromwell'*)

Configuration options for a MySQL instance

Parameters

- **url** – URL of the mysql instance (including port if not 3036)
- **username** – Username
- **password** – Password, not this is embedded into the Cromwell configuration (<output-dir>/janis/configuration/cromwell.conf)
- **dbname** – Database name to use, default 'cromwell'

Existing Cromwell instance

In addition to the previous `cromwell.url` method, you can also manage this through the command line. To configure Janis to submit to an existing Cromwell instance (eg: one already set up for the cloud), the CLI has a mechanism for setting the `cromwell_url`:

```
urlwithport="127.0.0.1:8000"
janis run --engine cromwell --cromwell-url $urlwithport hello
```

OR

'~/janis.conf'

```
engine: cromwell
cromwell:
  url: 127.0.0.1:8000
```

Overriding Cromwell JAR

In additional to the previous `cromwell.jar`, you can set the location of the Cromwell JAR through the environment variable `JANIS_CROMWELLJAR`:

```
export JANIS_CROMWELLJAR=/path/to/cromwell.jar
```

Recipes

Often between a number of different workflows, you have a set of inputs that you want to apply multiple times. For that, we have *recipes*.

Note: Recipes only match on the input name. Your input names _MUST_ be consistent through your pipelines for this concept to be useful.

class `janis_assistant.management.configuration.`**`JanisConfigurationRecipes`**(*type*)

`__init__`(*recipes: dict = None*, *paths: Union[str, List[str]] = None*, *directories: Union[str, List[str]] = None*)

Parameters

- **recipes** (`dict`) – A dictionary of input values, keyed by the recipe name.
- **paths** (`List[str]`) – a list of `*.yaml` files, where each path contains a dictionary of input values, keyed by the recipe name, similar to the previous recipes name.

- **directories** (*List[str]*) – a directory of `*.yaml` files, where the `*` is the recipe name.

For example, everytime I run the WGSGermlineGATK pipeline with `hg38`, I know I want to provide the same reference files. There are a few ways to configure this:

- Recipes: A dictionary of input values, keyed by the recipe name.
- Paths: a list of `*.yaml` files, where each path contains a dictionary of input values, keyed by the recipe name, similar to the previous recipes name.
- Directories: a directory of `*.yaml` files, where the `*` is the recipe name.

The examples below, encode the following information. When we use the `hg38` recipe, we want to provide an input value for reference as `/path/to/hg38/reference.fasta`, and input value for `type` as `hg38`. Similar for a second recipe for `hg19`.

Recipes dictionary

You can specify this recipe directly in your `janis.conf`:

```
recipes:
  recipes:
    hg38:
      reference: /path/to/hg38/reference.fasta
      type: hg38
    hg19:
      reference: /path/to/hg19/reference.fasta
      type: hg19
```

Recipes Paths

Or you could create a `myrecipes.yaml` with the contents:

```
hg38:
  reference: /path/to/hg38/reference.fasta
  type: hg38
hg19:
  reference: /path/to/hg19/reference.fasta
  type: hg19
```

And then instruct Janis to use this file in two ways:

1. In your `janis.conf` with:

   ```
   recipes:
     paths:
     - /path/to/myrecipes.yaml
   ```

2. OR, you can export the comma-separated environment variable:

   ```
   export JANIS_RECIPEPATHS="/path/to/myrecipes.yaml,/path/to/myrecipes2.yaml"
   ```

Recipe Directories

Create two files in a directory”

1. hg38.yaml:

   ```
   reference: /path/to/hg38/reference.fasta
   type: hg38
   ```

2. hg19.yaml:

   ```
   reference: /path/to/hg19/reference.fasta
   type: hg19
   ```

And similar to the paths, you can specify this directory in two ways:

1. In your janis.conf with:

```
recipes:
  directories:
  # /path/to/recipes has two files, hg38.yaml | hg19.yaml
  - /path/to/recipes/
```

2. OR, you can export the comma-separated environment variable:

```
export JANIS_RECIPEPATHS="/path/to/recipes/,/path/to/recipes2/"
```

Notifications

class janis_assistant.management.configuration.**JanisConfigurationNotifications**(*type*)

__init__(*email: str = None*, *from_email: str = 'janis-noreply@petermac.org'*, *mail_program: str = None*)

Parameters

- **email** – Email address to send status updates to
- **from_email** – (Default: janis-noreply@petermac.org)
- **mail_program** – Which mail program to use to send emails. A fully formatted email will be directed as stdin (eg: sendmail -t)

Environment

class janis_assistant.management.configuration.**JanisConfigurationEnvironment**(*type*)

__init__(*max_cores: int = None*, *max_memory: int = None*, *max_duration: int = None*)

Additional settings to configure a Janis environment. Currently, it mostly involves restricing resources (like cores, memory, duration) to fit within specific compute requirements. Notable, these values limit the requested values if they're a number. It doesn't currently limit this value if it's determined via an operator.

Parameters

- **max_cores** (*int*) – Limit the number of CPUs a job can request
- **max_memory** (*int*) – Limit the amount of memory (in GB) a job can request
- **max_duration** (*int*) – (Default: 86400) Limit the amount of time (in seconds) a job can request.

4.21 Configuring resources (CPU / Memory)

Sometimes you'll want to override the amount of resources a tool will get.

4.21.1 Hints

> There are changes to the tool docs coming to encapsulate this information.

Some tools are aware of hints, and can change their resources based on the hints you provide. For example, BWA MEM responds to the `captureType` hint (`targeted`, `exome`, `chromosome`, `30x`, `90x`).

4.21.2 Generating resources template

Sometimes you want complete custom control over which resources each of your tool, you can generate an inputs template for resources with the following:

```
$ janis inputs --resources BWAAligner

# bwamem_runtime_cpu: 16
# bwamem_runtime_disks: local-disk 60 SSD
# bwamem_runtime_memory: 16
# cutadapt_runtime_cpu: 5
# cutadapt_runtime_disks: local-disk 60 SSD
# cutadapt_runtime_memory: 4
# fastq: null
# reference: null
# sample_name: null
# sortsam_runtime_cpu: 1
# sortsam_runtime_disks: local-disk 60 SSD
# sortsam_runtime_memory: 8
```

4.21.3 Limitations

- You are unable to size a specific shard within a scatter. (See *Expressions in runtime attributes*)
- The only way to set the `disk` size for a tool is to generate the inputs template, and set a value.
- Currently (2020-03-20) you are unable to change the time limit of tasks

4.21.4 Upcoming work

> Leave an issue on GitHub (or comment on the linked issue) if you have comments.

Proposed: PMCC-BioinformaticsCore/janis-assistant#9

- Providing hints to get appropriately sized resources

  ```
  janis inputs [PROPOSED: --hint-captureType 30x] --resource WGSGermlineGATK
  ```

Proposed:

- Expressions in runtime attributes.

4.22 Call Caching

Call caching, or the ability for pipelines to use previous computed results. Each engine implements this separately, so this guide will assist you in setting up Janis, and how to configure the engines caching rules.

This feature can also be viewed as "resuming a workflow where you left off".

You may need to provide additional parameters when you run your workflow to ensure call-caching works. Please read this guide carefully.

4.22.1 Configuring Janis

You need to include the following line in your Janis Configuration:

```
# ...other options
call_caching_enabled: true
```

Strongly recommended:

- Use the `--development` run option as it ensures Cromwell will use the required database, and additionally sets the `--keep-intermediate-files` flag:
    - Remember, Janis will remove your execution directory (default: `<outputdir>/janis/execution`). And call caching only works if your intermediate files are immediately available.

CWLTool

No extra configuration should be required.

Cromwell

> More information about how Cromwell call-caching works below

You're running on a local or shared filesystem (including HPCs), we strongly recommend running Cromwell version 50 or higher, and to [use the `fingerprint` hashing strategy (Docs | PR):

```
cromwell:
  call_caching_method: fingerprint
```

You MUST additionally run Janis with the `--mysql` (recommended: `--development`) flag, Cromwell relies on a database for call-caching to work (unless you're running your own Cromwell server).

4.22.2 How does call-caching work

CWlTool

Needs further clarification.

Cromwell

> More information: Cromwell: Call Caching

More docs are in progress (here is our investigation), but specifically, Cromwell hashes all the components of your task:

- output count
- runtime attributes
- output expression
- input count
- backend name
- command template
- input - hash of the file, dependent on the filesystem (local, GCS, S3) - see more below.

and determines a total hash for the call you are going to make. Cromwell will then check in its database to see if it's made a call that exactly matches this hash, and if so uses the stored result.

If this *call* has NOT been cached, it will compute the result and then store it against the original hash for future use.

Input hash by filesystem

If you use a blob storage, like:

- Google Cloud Storage (GCS: `gs://`)
- Amazon Simple Storage Service (S3: `s3://`)

Cromwell can use the object id (the `etag`), as any modification to these files changes the object identifier.

The HTTP

Local filesystem

In a local filesystem, you don't have this object ID luxury, so there are a number of hashing strategies:

- `file`: md5 hash of the file content (bad for large files),
- `path`: computes an md5 hash of the file path (doesn't work for containers),
- `path+modtime`: computes md5 hash of file path + modtime (doesn't work for containers),
- `xxhash` (PROPOSED, > 50): Quicker hashing strategy for entire file.
- `fingerprint` (PROPOSED, > 50): Uses the `xxhash` strategy on (the first 10MB of file + mod time).

4.23 Common Workflow Language

Description about the CWL project, how some of the concepts map and limitations in the implementation.

4.23.1 Equivalents

- **How do I specify an Array from types?**

By wrapping the data_type by an Array, for example: `String -> Array(String)`. Nb: the Array type must be imported from janis_core.

- **What's the equivalent for ''InitialWorkDirRequirement''?**

You can add `localise_file=True` to your `ToolInput`. This is well defined for individual files in CWL and WDL. There is no equivalent for `writable`, though suggestions are welcome. Although the `localise_file` attribute is allowed for Array data types, the WDL translation will become disabled as this behaviour is not well defined.

From v1.1/CommandLineTool:

> *If the same File or Directory appears more than once in the InitialWorkDirRequirement listing, the implementation must choose exactly one value for path; how this value is chosen is undefined.*

- **What's the equivalent for ''EnvVarRequirement'' / how do I ensure environment variables are set in my execution environment?**

You can include the following code block within your CommandTool:

```
# Within CommandTool

def env_vars(self):
   return {
       "myvar1": InputSelector("myInput"),
       "myvar2": "constantvalue"
   }
```

4.24 Workflow Description Language

Description about the WDL project, how some of the concepts map and limitations in the implementation.

4.25 Collecting tool outputs

This guide will roughly walk you through the different ways of collecting outputs from a CommandTool. The bread and butter of this tutorial is the ToolOutput, and how you use various Selectors / Operators.

Examples:

- *Stderr / Stdout*
- *Wildcard / glob outputs*
- *Named outputs*

class janis.**ToolOutput**(*tag: str, output_type: Union[Type[Union[str, float, int, bool]], janis_core.types.data_types.DataType, Type[janis_core.types.data_types.DataType]], selector: Union[janis_core.operators.selectors.Selector, str, None] = None, presents_as: str = None, secondaries_present_as: Dict[str, str] = None, doc: Union[str, janis_core.tool.documentation.OutputDocumentation, None] = None, glob: Union[janis_core.operators.selectors.Selector, str, None] = None, _skip_output_quality_check=False*)

__init__(*tag: str, output_type: Union[Type[Union[str, float, int, bool]], janis_core.types.data_types.DataType, Type[janis_core.types.data_types.DataType]], selector: Union[janis_core.operators.selectors.Selector, str, None] = None, presents_as: str = None, secondaries_present_as: Dict[str, str] = None, doc: Union[str, janis_core.tool.documentation.OutputDocumentation, None] = None, glob: Union[janis_core.operators.selectors.Selector, str, None] = None, _skip_output_quality_check=False*)

A ToolOutput instructs the the engine how to collect an output and how it may be referenced in a workflow.

Parameters

- **tag** – The identifier of a output, must be unique in the inputs and outputs.
- **output_type** – The type of output that is being collected.
- **selector** – How to collect this output, can accept any `janis.Selector`.
- **glob** – (DEPRECATED) An alias for *selector*
- **doc** – Documentation on what the output is, used to generate docs.
- **_skip_output_quality_check** – DO NOT USE THIS PARAMETER, it's a scapegoat for parsing CWL ExpressionTools when an cwl.output.json is generated

You should note that there are key differences between how strings are coerced into Files / Directories in CWL and WDL.

- In WDL, a string is automatically coercible to a file, where the path is relative to the execution directory
- In CWL, a path is NOT automatically coercible, and instead a FILE object (`{path: "<path>", class: "File / Directory"}`) must be created. Janis shortcuts this instead, by inserting your strings as a globs, and letting CWL do this. There may be unintended side effects of this process.

4.25.1 Convention

We'll presume in this workflow that you've imported Janis like the following:

```
import janis_core as j
```

4.25.2 Examples

Stderr / Stdout

Collecting stdout and stderr can be done by simply annotating the types. This is functionally equivalent to type File, and using Stderr / Stdout as a selector:

```
outputs=[
    # stdout
    j.ToolOutput("out_stdout_1", j.Stdout()),
    j.ToolOutput("out_stdout_2", j.File(), selector=j.Stdout()),
    # stderr
    j.ToolOutput("out_stderr_1", j.Stderr()),
    j.ToolOutput("out_stderr_2", j.File(), selector=j.Stderr()),
]
```

Wildcard / glob outputs

If it’s not practical or impossible to determine the names of the outputs, you can use a *`janis.WildcardSelector`* to find all the files that match a particular pattern. This glob pattern is not transformed, and differences may occur between CWL / WDL depending on what glob syntax they use - please refer to their individual documentation for more information

- CWL: Globs
- WDL: Globs

You can use a glob in Janis with:

```
outputs=[
    j.ToolOutput("out_text_files", j.Array(j.File), selector=j.WildcardSelector("*.txt
↪")),
    # the next two are functionally equivalent
    j.ToolOutput("out_single_text_file_1", j.Array(j.File), selector=j.
↪WildcardSelector("*.txt", select_first=True)),
     j.ToolOutput("out_single_text_file_2", j.Array(j.File), selector=j.
↪WildcardSelector("*.txt")[0])
]
```

Roughly, this is translated to the following:

WDL:

```
Array[File] out_txt_files = glob("*.txt")
Array[File] out_single_txt_file_1 = glob("*.txt")[0]
Array[File] out_single_txt_file_2 = glob("*.txt")[0]
```

CWL:

Named outputs

Often we’ll use a string or a Filename generator to name an output of a tool. For example, `samtools sort` accepts an argument `-o` which is an output filename, on top of the regular `"bam"` input. We want our output to be of type Bam, so we’ll use a *`janis.Filename`* class, this accepts a few arguments: `prefix`, *suffix*‘ and `extension`, and will generate a filename based on these attributes.

We want our filename to be based on the input bam to keep consistency in our naming, so let’s choose the following attributes:

- `prefix` - will be the Bam, but want the file extension removed (this will automatically take the basename)
- `suffix` - *“.sorted”*‘
- `extension` - `.bam`

We can create the following ToolInput value to match this (we use a ToolInput as it means we could override it later):

```
ToolInput(
    "outputFilename",
    Filename(
        prefix=InputSelector("bam", remove_file_extension=True),
        suffix=".sorted",
        extension=".bam",
    ),
    position=1,     # Ensure it appears before the "bam" input
```

(continues on next page)

(continued from previous page)

```
    prefix="-o",     # Prefix, eg: '-o <output-filename>'
)
```

Then, on the ToolOutput, we can use the selector `selector=InputSelector("outputFilename")` to get this value. This results in the final tool:

```
SamtoolsSort_1_9_0 = CommandToolBuilder(
    tool="SamToolsSort",
    base_command=["samtools", "sort"],
    inputs=[
        ToolInput("bam", input_type=Bam(), position=2),
        ToolInput(
            "outputFilename",
            Filename(
                prefix=InputSelector("bam", remove_file_extension=True),
                suffix=".sorted",
                extension=".bam",
            ),
            position=1,
            prefix="-o",
        ),
    ],
    outputs=[ToolOutput("out", Bam(), selector=InputSelector("outputFilename"))],
    container="quay.io/biocontainers/samtools:1.9--h8571acd_11",
    version="1.9.0",
)
```

Looking at the relevant WDL:

```
task SamToolsSort {
  input {
    File bam
    String? outputFilename
  }
  command <<<
    set -e
    samtools sort \
      -o '~{select_first([outputFilename, "~{basename(bam, ".bam")}.sorted.bam"])}' \
      '~{bam}'
  >>>
  output {
    File out = select_first([outputFilename, "~{basename(bam, ".bam")}.sorted.bam"])
  }
}
```

And CWL:

```
class: CommandLineTool
id: SamToolsSort
baseCommand:
- samtools
- sort

inputs:
- id: bam
type: File
```

(continues on next page)

(continued from previous page)

```
inputBinding:
position: 2
- id: outputFilename
type:
- string
- 'null'
default: generated.sorted.bam
inputBinding:
prefix: -o
position: 1
valueFrom: $(inputs.bam.basename.replace(/.bam$/, "")).sorted.bam

outputs:
- id: out
type: File
outputBinding:
glob: $(inputs.bam.basename.replace(/.bam$/, "")).sorted.bam
```

4.26 Configuring Janis for HPCs

The Janis assistant currently implements Cromwell as an execution engine which supports HPCs.

The best way to use Janis and your HPC is to use one of the provided configurations. Currently, all of these configs use `singularity` to manage docker containers.

See a list of templates here.

4.26.1 Example: Slurm

- Template ID: `slurm_singularity`
- Documentation: `https://janis.readthedocs.io/en/latest/templates/slurm_singularity.html`
- CLI help: `janis init slurm_singularity --help`.

Example to configure this:

```
# This will write a janis.conf to your $HOME/.janis/janis.conf
$ janis init slurm_singularity

$ cat ~/.janis/janis.conf
# engine: cromwell
# notifications:
#   email: null
# template:
#   catch_slurm_errors: true
#   id: slurm_singularity
#   max_workflow_time: 20100
#   sbatch: sbatch
#   send_job_emails: false
```

Background mode

The slurm_singularity template constructs

4.27 Engine support

Janis currently support 2 engines:

- CWLTool
- Cromwell

Both of these engines have their limitations, and there will instructions in this document about those limitations and how they can be installed / configured.

4.27.1 CWLTool

CWLTool is the reference implementation for the Common Workflow Language (CWL). It's command line driven and can use Docker / Singularity for container driven processes. Unsurprisingly Janis transpiles your workflow to CWL to run with CWLTool.

Limitations

CWLTool only runs one tool at a time in serial. Even though there is a *parallel* mode, this isn't production stable, and Janis can't provide accurate metadata and progress tracking with this mode. CWLTool by itself also doesn't have the ability to submit to batch systems. With some configuration, the manager of Janis (that spins up CWLTool) can be submitted to a Batch System.

Janis determines the progress of your workflow by processing the stdout and converting it into the internal Workflow metadata models.

Known issues:

- Scattering on multiple fields with secondary files

Installation notes

The base CWLTool can be installed through Pip, see other installation methods here: GitHub: common-workflow-language.cwltool.

In addition **Node** should be installed, otherwise CWLTool is going to try and call out to Docker to run a node server. This doesn't always work in shared environments.

4.27.2 Cromwell

Cromwell is a workflow engine produced by the Broad Institute. It is very customisable for shared file systems (HPCs) and supports cloud computation through GCP (and supposed AWS).

Limitations:

CWL provides no pub-sub method for workflow notifications. Progress tracking is provided through requesting the metadata from the REST endpoint, and transforming this metadata into a recognised format for Janis. This metadata becomes quite large as the workflow grows, so is polled every 5-6 seconds.

- CWL: InitialWorkDirRequirement not returning new localized path when constructing command
- CWL output doesn't multiple workflow outputs that reference the same output of a tool

Installation notes:

Cromwell requires a Java 8 runtime environment, this must be loaded at submit for Janis to start Cromwell. Janis looks in the following places for a Cromwell jar:

- Environment (path): `JANIS_CROMWELLJAR`
- Configuration (path) : `cromwell.jarpath`
- ConfigDir (globbed): Searches inside the `configDir` for the pattern: `"cromwell-*.jar"` (default `~/.janis/`).

If no version of cromwell is found, then the latest version is downloaded from GitHub and placed in the config dir.

MySQL

Additionally, Janis (*expected* >= v0.8.0) will attempt to start a MySQL container for persistence, stability and eventually call caching of Cromwell. This is facilitated using Docker or Singularity. By default this is Docker, and can currently only be configured using an environment template.

The data files for MySQL are stored within your task execution directory, as a workflow gets larger these database files might grow in size. MySQL can be disabled using the `--no-mysql` flag on the CLI for `run`.

4.27.3 Unsupported engines

Here are a couple of other engines that we tried, but either haven't got them working yet or they don't quite fit our purposes.

Toil

Toil is primarily Python API driven workflow, it supports CWL through a bridge that uses CWLTool to parse and run the jobs. Although it is a good candidate for Janis, it's difficult to determine the progress of a workflow through the internal job store. It might be possible to generate the workflow in a particular way and rewrite the CWLToil bridge and parse the –stats output, but it was infeasible for this project.

- How to figure out progress using –stats
- toil-cwl-runner isn't assigning defaults for subworkflow.

MiniWDL

MiniWDL is an alternative engine for WDL support. miniWDL runs the exemplar WGS pipelines as expected on targeted panels, but no investigation has been completed how to obtain progress information or whether it can be configured to work with batch systems, the cloud or with singularity.

4.28 Updating Janis

Janis may seem like (an *organised*) chaos of individual Python packages, but they're individual to ensure proper separation of concerns, and allows us to provide bug fixes to each section separately.

The primary Janis module `janis-pipelines` (RELEASES page) contains a list of all the packages and their versions. But sometimes we make quick bug fixes to individual projects, without doing a big release of the whole project.

We recommend sticking to these major releases, you can update Janis to the latest major release with:

```
pip3 install --no-cache --upgrade janis-pipelines
```

4.28.1 Power users

Okay, so you're a power user and like to live on the edge. You need to decide now whether you like to live close to the cliff, or on the bleeding edge:

- Close-ish: Installing specific modules from Pip
- Bleeding edge: Installing individual projects from GitHub.

Installing from Pip

Janis projects follow a predictable naming structure:

> With one except, janis-assistant is called `janis-pipelines.runner` for legacy reasons.

```
janis-pipelines.<projectname>
```

You could for example update janis-core with:

```
pip3 install --no-cache --upgrade janis-pipelines.core
```

Bleeding edge: From GitHub

You should be warned that builds from GitHub are not always stable, and sometimes projects get released together as they contain inter-dependencies. We'd also recommend that you install from GitHub with the `--no-dependencies` flag.

Due diligence over, you can install from GitHub with the following line (replacing the GitHub link with the GitHub repo you're trying to install).

```
pip3 install --no-cache --upgrade --no-dependencies git+https://github.com/PMCC-
↪BioinformaticsCore/janis-<project>.git
```

4.29 Frequently asked questions

This document contains some common FAQ that may not have a place outside this documentation.

4.29.1 Janis Registry

- **Exception: Couldn't find tool: 'MyTool'**

To ensure your tool is correctly found by the registry (`JanisShed`), you must ensure that:

- The tool implements all abstract methods (can be initialised)
- Available within two levels of the janis-pipelines.tool extension root. This means that it needs to be within one additional import from the base `__init__`.

For example:

```
.
|-- __init__.py      # imports my_directory
|-- my_directory/
|    |-- __init__.py # imports tool from mytool.py
|    |-- mytool.py
```

If you're still having trouble, use `janis spider --trace mytool` to give you an indication of why your tool might be missing.

4.29.2 Command tools and the command line

- **Why is my input not being bound onto the command line?**

  You need to provide a `position` or `prefix` to a :class:`janis.ToolInput` to be bound on the command line.

- **How do I prefix each individual element in an array for my command line?**

  Set `prefix_applies_to_all_elements=True` on the :class:`janis.ToolInput`.

- **How do I make sure my file is in the execution directory? / How do I localise my file?**

  Set `localise_file=True` on the :class:`janis.ToolInput`. Although the `localise_file` attribute is allowed for Array data types, the WDL translation will become disabled as this behaviour is not well defined.

- **How do I include environment variables within my execution environment?**

  These are only available from within a CommandTool, and available by overriding the `env_vars(self)` method to return a dictionary of string to `Union[str, Selector]` key-value pairs.

- **Can a `janis.ToolInput` be marked as streamable?**

  Currently there's no support to mark ToolInput's as `streamable`. Although the `CWL specification`. does support marking inputs as streamable, WDL does not and, there is no engine support.

- **How can I manipulate an input value?**

  Janis only provides a limited way of manipulating input values, internally using a `StringFormatter`.

  Eg:

  - Initialisation
    - `StringFormatter("Hello, {name}", name=InputSelector("username"))`
  - Concatenation
    - `"Hello, " + InputSelector("username")`
    - `InputSelector("greeting") + StringFormatter(", {name}", name=InputSelector("username"))`

- **How do I specify an input multiple times on the command line?**
  - Create a `ToolInput` for the input you want to create.
    - Omit any binding options, ie NO `position` and NO `prefix`.
  - Create a `ToolArgument` with the value 'InputSelector("nameofyourinput") and the binding options:

  For example:

```
class MyTool(CommandTool):
    def inputs(self):
        return [ToolInput("myInput", String)]

    def arguments(selfs):
        return [
            ToolArgument(InputSelector("myInput"), position=1),
            ToolArgument("transformed-" + InputSelector("myInput"), position=2,␣
↪prefix="--name")
        ]
```

- **How do I ensure environment variables are set within my execution environment?**

  You can include the following block within your CommandTool:

```
# Within CommandTool

def env_vars(self):
    return {
        "myvar1": InputSelector("myInput"),
        "myvar2": "constantvalue"
    }
```

- **How do I make my generated filenames unique for a scatter or from different runs betweens tools?**

  Long story short, you can't.

  But there are a fun few reasons why that's currently the case:

  - The filenames are generated at transpile time for a tool wrapper. This means that tasks that use this tool will get the same filename, including if your workflow scatters over this task.
  - WDL doesn't really have a mechanism for achieving dynamic or generated components like this, and CWL was flaky at best.
  - Call caching in Cromwell relies on the command line being constructed, so the generated filenames currently break this, and a purely randomly generated filename would breat this further.
  - We have cascaded filename components on the roadmap, so your filename can be built from a collection of inputs (sort of possible anyway).

4.29.3 Running workflows

- **How do I detect a workflow's status from the command line?**

  You can request metadata for a workflow You can use the following bash command to detect the status.

```
janis metadata <dir / wid> |grep status|tr -s ' '|cut -d ' ' -f 2
```

  Note that if Janis is killed suddenly, it might not have enough time to mark the workflow as failed or terminated. It's worth checking the last_updated_time, if a workflow hasn't been updated in a couple of hours, it's likely Janis has been killed.

  The metadata will have the following keys:

  - https://github.com/PMCC-BioinformaticsCore/janis-assistant/blob/master/janis_assistant/data/enums/workflowmetadatakey

  And the workflow status can have the following states:

  - https://github.com/PMCC-BioinformaticsCore/janis-assistant/blob/master/janis_assistant/data/enums/taskstatus.py

- **How can I override a specific tools' resources, like memory / ram, CPUs, or runtime (coming soon)?**

  Long story short, lookup the override key with:

  ```
  janis inputs --resources <workflow>
  ```

  It looks like `yourworkflow_embeddedtool_runtime_memory: 8`. More information: Configuring resources (CPU / Memory).

4.29.4 Containers

- **How do I override the container being used?**

  You can override the container being used by specifying a `container-override`, there are two ways to do this:

  - CLI with the syntax: `janis [translate|run] --container-override 'MyTool=myorganisation/mytool:0.10.0' mytool`
  - API: include container override dictionary: `mytool.translate("wdl", container_override={"mytool": "myorganisation/mytool:0.10.0"})`, the dictionary has structure: `{toolId: container}`. The toolId can be an asterisk `*` to override all tools' containers.

4.30 Common errors

From time to time you'll come across error messages in Janis, and hopefully they give you a good indication of what's happened, but here we'll address some.

If you've got a strange error message and don't know what's happening, post a message on Gitter or raise an issue on GitHub!

4.30.1 Command line positions

Command line positioning can be a little confusing. When running or translating you should follow this format:

```
janis run [run-arguments] yourworkflow.py [workflow-inputs]
```

Where

- `run-arguments` are options like `--inputs`, `--recipe`, `--hint`, `--engine`, `--stay-connected`, etc.
- `workflow-inputs` are additional inputs you

Errors:

- **There were unrecognised inputs provided to the tool "yourtool", keys: engine, recipe, hint, otherkey**

  1. In the command line, you might have accidentally placed a run argument in the workflow inputs section. Eg: :

     - Wrong: `janis run yourtool --engine cwltool`
     - Correct: `janis run --engine cwltool yourtool`

  2. You might have also included workflow inputs that yourtool did not recognise. Please ensure that yourtool accepts all the unrecognised keys.

4.30.2 Finding files

Janis looks in a number of places outside just your current directory to find certain types of files. Janis will look in the following places for your file:

- The full path of the file (if relevant)
- Current directory
- `$JANIS_SEARCHPATH` (if defined in the environment).

If your path starts with `https://` or `http//`, the file will be downloaded to a cache in the configuration directory (default: `~/.janis/cache`) and will not go through the search path procedure.

You can run your command with `janis -d yourcommand` and look for the following lines which tell you how Janis is searching for files:

```
[DEBUG]: Searching for a file called 'hellop'
[DEBUG]: Searching for file 'hellop' in the cwd, '/path/to/cwd/'
[DEBUG]: Attempting to get search path $JANIS_SEARCHPATH from environment variables
[DEBUG]: Got value for env JANIS_SEARCHPATH '/path/to/janis/', searching for file 'hellop' here.
[DEBUG]: Couldn't find a file with filename 'hellop' in any of the following: full path, current working directory (/path/to/cwd/) or the search path.
[DEBUG]: Setting CONSOLE_LEVEL to None while traversing modules
[DEBUG]: Restoring CONSOLE_LEVEL to DEBUG now that Janis shed has been hydrated
# It's now searching the registry and will fail after it can't find it.
```

Errors:

- **Exception: Couldn't find workflow with name: myworkflow**

  This might occur during a `run` or `translate`. Janis couldn't find `myworkflow` in any of the usual spots, nor in the tool registry.

  - If you're referencing a Python file (eg: `myworkflow.py`), try giving the full path to the file or ensure that it's correctly in your search paths.
  - If you're referencing a tool that should be in the store, check the spelling of the tool. The name you reference needs to be the ToolId (under `def tool(): return "toolid"`).
  - If you're referencing a tool that you're putting into the store, ensure there are no syntax / runtime errors in the Python file. You can double check this a few ways:
    - Running `janis translate /path/to/your/file.py wdl` and ensure that runs correctly.
    - Adding an `if __name__ == "__main__": YourWorkflowClass().translate("wdl")` to the bottom of your python file, then running `python /path/to/your/file.py` and checking that it translates correctly.

- **FileNotFoundError: Couldn't find inputs file: myinput.yml**

  This is likely because your inputs file `myinput.yml` couldn't be found in the search path. Try giving the full path to your file.

- **Unrecognised python file when getting workflow / command tool: hello.py :: $PYTHONERROR**

  This error results in a couple of ways:

  - There was a problem when parsing your file (`hello.py`). The error (given by `$PYTHONERROR`) was returned by Python when trying to interpret your file and get the workflow out. This could be syntactic or runtime error in your file.
  - You were looking a tool up in the registry, but you had a file / folder in your search path that caused a clash. You can include the `--registry` parameter after the `run` to only look up the tool in the registry.

- **There was more than one workflow (3) detected in 'hel.py' (please specify the workflow to use via the `--name` parameter, this name must be the name of the variable or the class name and not the workflowId). Detected tokens: 'HelloWorkflow' (HelloWorkflow), 'HelloWorkflow2' (HelloWorkflow2), 'w' (HelloWorkflow)**

  When looking for workflows and command tools within the file `myworkflow.py`, Janis found more than one workflow / command tool that could be run, in this case there were 3:

  - *HelloWorkflow* (`HelloWorkflow`)
  - *HelloWorkflow2* (`HelloWorkflow2`)
  - *w* (`HelloWorkflow`)

  You need to specify the `--name` parameter with one of these ids (`HelloWorkflow`, `HelloWorkflow2` or `w`) to run / translate / etc.

- **ValueError: There were errors in 2 inputs: {'field1': 'value was null', 'field2': 'other reason'}**

  One or more of your inputs when running were invalid. The dictionary gives the field that was invalid (key) and the reason Janis thinks your value was invalid (value).

4.30.3 Containers

- **Exception: The tool 'MyTool' did not have a container. Although not recommended, Janis can export empty docker containers with the parameter `allow_empty_container=True` or `--allow-empty-container`**

  One of the tools that you are trying to run or translate did not have a container attached to this. In Janis, we strongly encourage the use of containers however it's not always possible.

  To allow a tool to be translated correctly, there are two mechanisms for doing so:

  - Translate without a container
    - CLI: Include the `--allow-empty-container` flag, eg: `janis [translate|run] --allow-empty-container mytool`
    - API: Include `allow_empty_container=True`, `mytool.translate("cwl", allow_empty_container=True)`.
  - Override the container
    - CLI with the syntax: `janis [translate|run] --container-override 'MyTool=myorganisation/mytool:0.10.0' mytool`

    - API: include container override dictionary: `mytool.translate("wdl", container_override={"mytool": "myorganisation/mytool:0.10.0"})`, the dictionary has structure: `{toolId: container}`. The toolId can be an asterisk `*` to override all tools' containers.

4.31 Parsing CWL

Common Workflow Language (CWL) is a common transport and execution format for workflows. Janis translates to CWL, and now Janis can *partially* translate from CWL.

4.31.1 Quickstart

Make sure you have `janis-pipelines >= 0.11.1` AND `janis-pipelines.core >= 0.11.3` installed.

Note, we're using `janisdk` NOT the regular `janis` CLI:

```
janisdk fromcwl <yourworkflow.cwl>
```

If this translates correctly, this will produce python source code from your workflow.

4.31.2 Case study

Result: https://github.com/PMCC-BioinformaticsCore/janis-pipelines/tree/master/janis_pipelines/kidsfirst

Converting Kids-First pipelines:

- alignment workflow
- rnaseq workflow
- somatic workflow
- joint genotyping

Clone one of the workflows to a location. Some things to watch out for:

- Ensure that no tool ids clash (eg: workflow.id, commandlinetool.id), Janis relies on these being unique IDS as they get used to generate variable names.
- Expressions that don't translate, these get logged as warnings.

For example, when translating the `kfdrc-rnaseq-workflow.cwl`, we find the following warnings:

```
$ janisdk fromcwl /Users/franklinmichael/source/kf-rnaseq-workflow/workflow/kfdrc-
↪rnaseq-workflow.cwl

[INFO]: Loading CWL file: /Users/franklinmichael/source/kf-rnaseq-workflow/workflow/
↪kfdrc-rnaseq-workflow.cwl
[WARN]: Couldn't translate javascript token, will use the placeholder '<expr>
↪"*TRIMMED." + inputs.readFilesIn1</expr>'
[WARN]: Couldn't translate javascript token, will use the placeholder '<expr>inputs.
↪strand ? inputs.strand == "default" ? "" : "--"+inputs.strand : ""</expr>'
[WARN]: Couldn't translate javascript token, will use the placeholder '<expr>inputs.
↪reads2 ? inputs.reads1.path+" "+inputs.reads2.path : "--single -l "+inputs.avg_frag_
↪len+" -s "+inputs.std_dev+" "+inputs.reads1</expr>'
```

(continues on next page)

(continued from previous page)

```
[WARN]: Couldn't translate javascript token, will use the placeholder '<expr>inputs.
↪chimeric_sam_out.nameroot</expr>'
[WARN]: Couldn't translate javascript token, will use the placeholder '<expr>inputs.
↪unsorted_bam.nameroot</expr>'
[WARN]: Couldn't translate javascript token, will use the placeholder '<expr>inputs.
↪chimeric_sam_out.nameroot</expr>'
[WARN]: Couldn't translate javascript token, will use the placeholder '<expr>inputs.
↪readFilesIn2 ? inputs.readFilesIn2.path : ''</expr>'
[WARN]: Couldn't translate javascript token, will use the placeholder '<expr>inputs.
↪genomeDir.nameroot.split('.'</expr>'
[WARN]: Expression tools aren't well converted to Janis as they rely on unimplemented␣
↪functionality: file:///Users/franklinmichael/source/kf-rnaseq-workflow/tools/
↪expression_parse_strand_param.cwl#expression_strand_params```
```

4.31.3 Limitations

- Can only convert very basic Javascript expressions
- `when` conditionals aren't supported yet (we can't translate the expressions very well)
- Some requirements may not be faithfully represented
- Expression tools get translated, but they require extra Janis features to be implemented to work:
  - Add ReadJsonOperator
  - Allow Stdout / stderr to better participate in operations

Expressions are one of the trickiest bits of the translation, at the moment we can only parse some extremely basic expressions, for example:

- `${ return 800 }`: Literal value `800`
- `$(inputs.my_input)`: `InputSelector("my_input")`
- `$(inputs.my_input.basename)`: `BasenameOperator(InputSelector("my_input"))`
- `$(inputs.my_input.size)`: `FileSizeOperator(InputSelector("my_input"))`
- `$(inputs.my_input.contents)`: `ReadContents(InputSelector("my_input"))`

Developer notes

Source code: `janis_core/ingestion/fromcwl.py`

Expressions

Expressions are parsed using a combination of regular expressions and python string matching, there's nothing super clever there.

Tests: `janis_core/tests/test_from_cwl.py`

4.32 Development

> This project is work-in-progress and is still in developments. Although we welcome contributions, due to the immature state of this project we recommend raising issues through the Github issues page.

4.32.1 Introduction

This page has a collection of notes on the development of Janis.

Testing

> *Further information*: Testing

Within Janis there is a suite of code-level tests. High quality tests allow fewer bugs to make it out to a production environment and gives you a place to test your function.

All of the tests are placed in: `janis/tests/test_*.py`. You can create a file in that same directory with tests for your changes.

4.32.2 Running tests

In the terminal you can run the tests and generate a code coverage report with the following bash command.

```
nosetests -w janis --with-coverage --cover-package=janis
```

Releasing

> *Further Information*: Releasing

Releasing is automatic! After you've run the tests, simply increment the version number in `setup.py` (with resepect to SemVer), and tag that commit with the same version identifier:

```
git commit -m "Tag for v0.x.x release"
git tag -a "v0.x.x" -m "Tag message"
git push origin v0.x.x
```

Logger

This project uses a custom logger that can be imported from the root of Janis:

```
from janis import Logger
```

- `Logger.mute()` stops any output from reaching the console until `unmute()` is called. This does not affect logging to disk.
- `Logger.unmute()` restores the logger's ability to write to the console if it has been muted. It will restore to the same *volume* before it was muted.
- Logging functions:
  - `.log(str)` - `LogLevel.DEBUG`
  - `.info(str)` - `LogLevel.INFO`
  - `.warn(str)` - `LogLevel.WARNING`
  - `.critical(str)` - `LogLevel.CRITICAL`

  - `.log_exec(exception)` - `Loglevel.CRITICAL` - Prepares exception in standard format

Adding a translation

There are a few things that are super helpful to know before you add a new translation. You should be familiar with Janis' representation of a workflow.

> It's recommended you use an intermediary library (similar to cwlgen or python-wdlgen) to manage the representation. This allows you to focus on mapping concepts over syntax.

4.32.3 Concepts

There are a few things you'll need to keep in mind when you add a

translation, you should understand at least these workflow and janis concepts: | - inputs - outputs - steps + tools - secondary files - command line binding of tools - position, quoted - selectors - input selectors - wildcard glob selectors - cpu and memory selectors - propagated resource overrides

4.32.4 Translating

Add a file to the `janis/translation` with your new language's name. Create a class called: `$LangTranslator` that should inherit from `TranslatorBase`, and provide implementations for those methods.

Please keep your function sizes small and direct, and then write unit tests to cover each component of the translation and then an integration test of the whole translation on the related workflows.

You can find these in `/janis/tests/test_translation_*.py`)

4.33 Contributing

We'd love to accept community contributions, especially to add new tools to:

- Janis-bioinformatics
- Janis-unix
- Or other tool suites!

There are a couple of best standard we'd love to follow when contributing to Janis.

4.33.1 Documentation

There are a couple of ways that can you provide documentation to your tools:

Tool and workflow inputs

There's a `doc` parameter that can be filled when creating a `ToolInput` or `Workflow.input` that is used to drive the Janis documentation and table of inputs for a tool.

```
ToolInput(
        "inputId",
        DataType,
        doc="<Explain what this input does>",
        **kwargs
)
```

```
w = WorkflowBuilder()
w.input(
        "inputId",
        DataType,
        doc="<Explain what this input does>",
        **kwargs
)
```

Tool documentation

This differs if you're using a class based tool, or using of the builders.

Class based

A `CommandTool` and `Workflow` both have a `bind_metadata` method that you can return a `ToolMetadata` or `WorkflowMetadata` from respectively.

CommandTool:

```
def bind_metadata(self):
        from datetime import date

        return ToolMetadata(
                contributors=["<Add your name here>"],
                dateCreated=date(2018, 12, 24),
                dateUpdated=date(2020),
                institution="<who produces the tool>",
                keywords=["list", "of", "keywords"],
                documentationUrl="<url to original documentation>",
                short_documentation="Short subtitle of tool",
                documentation="""Extensive tool documentation here """.strip(),
                doi=None,
                citation=None,
  )
```

Workflow

```
def bind_metadata(self):
        from datetime import date

        return ToolMetadata(
                contributors=["<Add your name here>"],
                dateCreated=date(2018, 12, 24),
                dateUpdated=date(2020),
                institution="<who produces the tool>",
                keywords=["list", "of", "keywords"],
                documentationUrl="<url to original documentation>",
```

(continues on next page)

(continued from previous page)

```
                short_documentation="Short subtitle of tool",
                documentation="""Extensive tool documentation here """.strip(),
                doi=None,
                citation=None,
  ).
```

Builders

The builders have a parameter called `metadata` which takes a `ToolMetadata` or `WorkflowMetadata` for a `CommandToolBuilder` or `WorkflowBuilder` respectively:

Example

```
wf = WorkflowBuilder(
    "workflowId",
    **kwargs,
    metadata=WorkflowMetadata(
        **meta_kwargs
    ),
)

# OR

clt = CommandTooBuilder(
    "commandtoolId",
    **kwargs,
    metadata=ToolMetadata(
        **meta_kwargs
    ),
)
```

4.33.2 Code formatting

Janis uses the opinionated Black Python code formatter. It ensures that there are no personal opinions about how to format code, and anecdotally saves a lot of mental power from not thinking about this.

The projects include a `pyproject.toml` , and it's a git pre-commit dependency.

Installation

You can install Black (+ pre-commit) inside a `janis-*` directory with:

```
pip3 install -r requirements.txt
```

You can manually run black with:

```
black {source_file_or_directory}
```

Editor integration

> See more: https://github.com/psf/black

We'd recommend configuring Black to run on save within your editor with the following guides:

- PyCharm / IntelliJ IDEA
- Vim
- VSCode + Python extension, simplified instructions:
  - Install the [extension]
  - Within VSCode, go to your settings.json
    - Command + Shift + P
    - (type) Preferences: Open Settings (JSON)
  - Add the following lines:

```
    "python.formatting.provider": "black",
    "editor.formatOnSave": true
```

4.34 Testing

Build Status codecov

Testing is an important part Software Development, and Janis is no exception. Within the core `janis` project, there is a suite of code-level tests. High quality tests allow developers to be more confident in their code, and will decrease the amount of bugs found in production code.

For every issue, it's worth looking at is there a tests than could have caught the error we're facing, that way we can stop someone reintroducing the bug (regression).

4.34.1 Where to place tests

For each new logical section, you should create a file with a new set of tests. All of the tests are placed in: `janis/tests/test_*.py`.

4.34.2 How to run tests

In the terminal you can run:

```
nosetests -w janis
```

Alternatively if you use Pycharm, you can right-click the `janis/tests/` folder, and click `Run 'Unittests in tests'`.

Run unit tests

4.34.3 Code Coverage

codecov

Code coverage isn't a great metric for evaluating how good our tests are, though it does give an indication on whether more tests are needed.

Code coverage is automatically generated by the build server and uploaded to `codecov` (that's where the badge comes from).

You can generate codecov locally by running:

```
nosetests -w janis --with-coverage --cover-package=janis
```

4.34.4 Code Quality

A style guide hasn't been produced and hence a list of practices are not enforced, however it's best to follow the PEP guidelines. You can use `mypy` to run some basic tests, we like to keep these suggestions minimised:

```
mypy janis
# OR
mypy --ignore-missing-imports .
```

> Ignore missing imports stops a lot of extraneous errors from third-party modules from showing up

Type annotations

The project uses type annotations to improve the clarity of code. Using tools such as `mypy` allows us to catch typing errors that might not have been easier caught by other static analysers.

For this reason the project is pegged to at least Python 3.6.

Why might a type system be good?

Although not strictly related to Python, JS has some super fun issues with types:

```
1 + 1                 // 2
'1' + 1                // '11'
1 + '1'                // 2
let a = '1'
a -= 1                // 2
```

Some of these examples come from this discussion of a postmortem from airbnb.

4.35 Releasing Janis

Build Status PyPI version

Releasing janis is straight forward. Decide on a logical set of changes to include, this will

4.35.1 Versioning

Janis follows the `SemVer` versioning system:

1. MAJOR version when you make incompatible API changes,
2. MINOR version when you add functionality in a backwards-compatible manner, and
3. PATCH version when you make backwards-compatible bug fixes.

Before a new release, you should update the version tag in `janis/__init__.py` so the produced python package is

4.35.2 Tagging and Building

> Before you tag, make sure you've incremented the `__version__` flag per the previous section.

You can tag your commit for release by running the following bash command:

```
git commit -m "Tag for v0.x.x release"
git tag -a "v0.x.x" -m "Tag message"
git push origin v0.x.x
```

The final statement will push the tag to Github, where Travis will automatically pick up commit, build and deploy.

> You can check the build status and history on `travisci/janis`. You can also check `PyPi` to confirm the build has been released.

Failing builds for tags

If your tag fails to build, you'll need to:

1. Delete the tag from Github: https://github.com/PMCC-BioinformaticsCore/janis/releases/tag/vX.X.X
2. Delete the tag from your local: `git tag -d 'vX.X.X'`
3. Retag and push per *Tagging and Building*

4.35.3 Github release notes

You should update the release notes in the `CHANGELOG.md` where you'll need to create a `##` section with three bits of information:

1. Release commit hash (*eg*: `bc7f8ad`)
2. Github link to commits comparison: `https://github.com/PMCC-BioinformaticsCore/janis/compare/$prev...$new`
   - Where `$prev` and `$new` is your old, and release commit hash respectively.
3. Release notes: a summary of changes since the last release

CHAPTER 5

Indices and tables

- genindex
- search
- modindex

Index

Symbols

__init__() (*janis.CommandToolBuilder method*), 76
__init__() (*janis.InputDocumentation method*), 97
__init__() (*janis.ScatterDescription method*), 94
__init__() (*janis.ToolArgument method*), 81
__init__() (*janis.ToolInput method*), 80
__init__() (*janis.ToolOutput method*), 81, 118
__init__() (*janis_assistant.management.configuration.JanisConfiguration_ method*), 109
__init__() (*janis_assistant.management.configuration.JanisConfigurationCromwell method*), 111
__init__() (*janis_assistant.management.configuration.JanisConfigurationEnvironment method*), 114
__init__() (*janis_assistant.management.configuration.JanisConfigurationNotifications method*), 114
__init__() (*janis_assistant.management.configuration.JanisConfigurationRecipes method*), 112
__init__() (*janis_assistant.management.configuration.JanisConfigurationTemplate method*), 110
__init__() (*janis_assistant.management.configuration.MySqlInstanceConfig method*), 111

B

bind_metadata() (*janis.Workflow method*), 75
Boolean (*class in janis*), 87

C

code_block() (*janis.PythonTool static method*), 82
CommandTool (*class in janis*), 76
CommandToolBuilder (*class in janis*), 76
config_dir (*janis_assistant.management.envvariables.EnvVariables attribute*), 108
config_path (*janis_assistant.management.envvariables.EnvVariables attribute*), 109
constructor() (*janis.Workflow method*), 75
cromwelljar (*janis_assistant.management.envvariables.EnvVariables attribute*), 109

D

default_template (*janis_assistant.management.envvariables.EnvVariables attribute*), 109

E

EnvVariables (*class in janis_assistant.management.envvariables*), 108
exec_dir (*janis_assistant.management.envvariables.EnvVariables attribute*), 109

F

File (*class in janis*), 86
Filename (*class in janis*), 87
Float (*class in janis*), 87
friendly_name() (*janis.Workflow method*), 74

I

id() (*janis.Workflow method*), 74
input() (*janis.Workflow method*), 72
InputDocumentation (*class in janis*), 97
InputSelector (*class in janis*), 92
Int (*class in janis*), 86

J

JanisConfiguration (*class in janis_assistant.management.configuration*), 109
JanisConfigurationCromwell (*class in janis_assistant.management.configuration*), 111
JanisConfigurationEnvironment (*class in janis_assistant.management.configuration*), 114
JanisConfigurationNotifications (*class in janis_assistant.management.configuration*), 114
JanisConfigurationRecipes (*class in janis_assistant.management.configuration*), 112

JanisConfigurationTemplate (*class in janis_assistant.management.configuration*), 110

M

MySqlInstanceConfig (*class in janis_assistant.management.configuration*), 111

O

output() (*janis.Workflow method*), 73

output_dir (*janis_assistant.management.envvariables.EnvVariables attribute*), 109

P

PythonTool (*class in janis*), 82

R

recipe_directory (*janis_assistant.management.envvariables.EnvVariables attribute*), 109

recipe_paths (*janis_assistant.management.envvariables.EnvVariables attribute*), 109

S

ScatterDescription (*class in janis*), 94

ScatterMethods (*in module janis*), 94

search_path (*janis_assistant.management.envvariables.EnvVariables attribute*), 109

step() (*janis.Workflow method*), 72

String (*class in janis*), 86

StringFormatter (*class in janis*), 93

T

ToolArgument (*class in janis*), 81

ToolInput (*class in janis*), 80

ToolOutput (*class in janis*), 81, 118

translate() (*janis.Workflow method*), 71

TTestCase (*class in janis_core.tool.test_classes*), 107

TTestExpectedOutput (*class in janis_core.tool.test_classes*), 107

TTestPreprocessor (*class in janis_core.tool.test_classes*), 107

W

WildcardSelector (*class in janis*), 93

Workflow (*class in janis*), 71

WorkflowBuilder (*class in janis*), 71