INSTITUTIONEN FÖR SVENSKA,
FLERSPRÅKIGHET OCH SPRÅKTEKNOLOGI

GÖTEBORGS
UNIVERSITET

GU-ISS-2022-01

Sparv 5  Developer's Guide

Martin Hammarstedt, Anne Schumacher, Lars Borin, Markus Forsberg

Forskningsrapporter från Institutionen för svenska, flerspråkighet
och språkteknologi, Göteborgs universitet
Research Reports from the Department of Swedish, Multilingualism, Language Technology

ISSN 1401-5919

Contents

1 General Concepts

This section will give a brief overview of how Sparv modules work and introduce some general concepts. More details are provided in the following chapters.

The Sparv Pipeline is comprised of some core functionality and many different modules containing Sparv functions that serve different purposes like reading and parsing source files, building or downloading models, producing annotations and producing output files that contain the source text and annotations. All of these modules (i.e. the code inside the `sparv/modules` directory) are replacable. A Sparv function is decorated with a special decorator that tells Sparv what purpose it serves. A function's parameters hold information about what input is needed in order to run the function and what output is produced by it. The Sparv core automatically finds all decorated functions, scans their parameters and builds a registry for what modules are available and how they depend on each other.

1.1 Annotations

The most common Sparv function is the annotator which produces one or more annotations. An annotation consists of spans that hold information about what text positions it covers and an optional attribute for each span. For example, a function that segments a text into tokens produces a span annotation that tells us where each token begins and ends in the source text. A function that produces part-of-speech tags on the other hand would rely on the token spans produced by another function and just add an attribute for each token span, telling us whether this token is a noun or a verb or something else.

Annotations are referred to by their internal names which follow a strict naming syntax. The name of a span annotation starts with the name of the module that it is produced by, followed by a dot, followed by an arbitrary name consisting of lowercase ASCII letters and underscores. The token span annotation produced by the `segment` module for example is called `segment.token`. The name of an attribute annotation follows the same rules, except that it is prefixed with the name of the span annotation that it is adding attributes to, and a colon. So the part-of-speech annotation produced by the `stanza` module is called `segment.token:stanza.pos` because it adds part-of-speech attributes to the `segment.token` span annotation.

1.2 Dependencies

Some Sparv functions may require annotations from other functions before they can be run. These dependencies are expressed in the function arguments. By using special Sparv classes as default arguments in a function's signature the central Sparv registry can automatically keep track of what annotations can be produced by what function and in what order things need to be run. These dependencies can either be described in a module specific manner or in a more abstact way. For example, an annotator producing word base forms (or lemmas) may depend on a part-of-speech annotation with a specific tagset and therefore this annotator might define that its input needs to be an annotation produced by a specific module. A part-of-speech tagger on the other hand usually needs word segments as input, and it probably does not matter exactly what module produces these segments. In this case the dependency can be expressed with an abstract annotation class.

1.3 Annotation Classes

When describing dependencies to other annotations one can make use of annotation classes which are denoted by angle brackets (e.g. `<token>`, `<token:word>`). Annotation classes are used to create abstract instances for common annotations such as tokens, sentences and text units. They simplify dependencies between annotation modules and increase the flexibility of the annotation pipeline. Many annotation modules need tokenised text as input, but they might not care about what tokeniser is being used. So instead of telling a module that it needs tokens produced by another specific module we can tell it to take the class `<token>` as input. In the corpus configuration we can then set `classes.token` to `segment.token` which tells Sparv that `<token>` refers to output produced by the `segment` module.

Annotation classes are valid across all modules and may be used wherever you see fit. There is no closed set of annotation classes and each module can invent its own classes if desired. Within a corpus directory all existing classes can be listed with the `sparv classes` command.

2 Writing Sparv Plugins

The Sparv Pipeline is comprised of different modules like importers, annotators and exporters. Although many modules are shipped with the main Sparv package none of these modules are hard-coded into the Sparv Pipeline and therefore it can easily be extended with plugins. A plugin is a Sparv module that is not part of the main Sparv package. Writing a plugin is the recommended way of adding a new module to Sparv.

> **Note:** When writing a plugin please always prefix your Python package with a namespace followed by an underscore to mark which organisation or developer the plugin belongs to. This is necessary to avoid clashes in package names and obligatory plugin namespaces will be enforced in the future. In the example below we used the prefix "sbx_" (for Språkbanken Text).

When writing your first plugin we recommend that you take a look at the Sparv plugin template. The template contains an example of a small annotation module that converts tokens to uppercase. We will use this template in the examples below.

2.1 Plugin Structure

This is what a typical structure of a plugin may look like:

```
sparv-sbx-uppercase/
├── sbx_uppercase
│   ├── uppercase.py
│   └── __init__.py
├── LICENSE
├── README.md
└── setup.py
```

In the above example the `sbx_uppercase` directory is a Sparv module containing the module code in `uppercase.py` and the mandatory init file `__init__.py`. The setup file `setup.py` in the root directory is needed in order to install the plugin.

The readme and license files are not strictly necessary for the plugin to work but we strongly recommend that you include these if you want to publish your plugin.

2.2 Setup File

The `setup.py` is needed in order to install a plugin and connect it to the Sparv Pipeline. Here is a minimal example of a setup file (taken from the Sparv plugin template):

```
import setuptools

setuptools.setup(
    name="sparv-sbx-uppercase",
    version="0.1",
    description="Uppercase converter (example plug-in for Sparv)",
    license="MIT",
    packages=["sbx_uppercase"],
    python_requires=">=3.6",
    install_requires=["sparv-pipeline>=4,<5"],
    entry_points={"sparv.plugin": ["sbx_uppercase = sbx_uppercase"]}
)
```

Make sure to include the name of your module (i.e. the directory containing the Sparv code) in `packages`. You also need to make sure that there is a `sparv.plugin` entry point in `entry_points` that points to your module.

We strongly encourage you to also include the fields `author` and `author_email`.

For more information about Python setup scripts check the distutils documentation.

2.3 Init File

Each Sparv module must contain a Python init file (`__init__.py`). Without the init file Sparv will not be able to register the module. The Python scripts containing decorated Sparv functions should be imported here. Module-specific configuration parameters may also be declared in this file. Furthermore, you should provide a short description (one sentence) for your module which will be displayed to the user when running the `sparv modules` command. The description is provided either as an argument to `__description__` or as a docstring. In the example below we use both, but only one of them is necessary. If both exist, the value of `__description__` is displayed in the `sparv modules` command.

Example of an `__init__.py` file:

```
"""Example for a Sparv annotator that converts tokens to uppercase."""

# from sparv.api import Config

from . import uppercase

# __config__ = [
#     Config("uppercase.some_setting", "some_default_value", description="
    Description for this setting")
# ]

__description__ = "Example for a Sparv annotator that converts tokens to
    uppercase."
```

2.4 Module Code

A Sparv module is a Python package containing at least one Python script that imports Sparv classes (and util functions if needed) which are used for describing dependencies to other entities (e.g. annotations or models) handled or created by the pipeline. Here is the code for or uppercase example (taken from the Sparv plugin template:

```
from sparv.api import Annotation, Output, annotator

@annotator("Convert every word to uppercase.")
def uppercase(word: Annotation = Annotation("<token:word>"),
              out: Output = Output("<token>:sbx_uppercase.upper")):
    """Convert to uppercase."""
    out.write([val.upper() for val in word.read()])
```

In this script we import two classes from Sparv (`Annotation` and `Output`) and the `annotator` decorator. Please note that nothing should be imported from the Sparv code unless it is directly available from the sparv.api package (i.e. `from sparv.api import ...`). Any other sub-packages (like `sparv.core`) are for internal use only, and are subject to change without notice.

Our `uppercase` function is decorated with `@annotator` which tells Sparv that this function can be used to produce one or more annotations. The first argument in the decorator is its description which is used for displaying help texts in the CLI (e.g. when running `sparv modules`).

The function's relation to other pipeline components is described by its signature. The function arguments contain type hints to the Sparv classes `Annotation` and `Output` which indicate what dependencies (e.g. annotations, models or config variables) must be satisfied before the function can do its job, and what it will produce. In this example Sparv will make sure that a word annotation exists before it will attempt to call the `uppercase` function, because it knows that `word` is an input since it is of type `Annotation`. It also knows that the function produces the output annotation `<token>:sbx_uppercase .upper`, so if any other module would request this annotation as input, it will run `uppercase` prior to calling that module.

A function decorated with a Sparv decorator should never be actively called by you or by another decorated function. When running Sparv through the CLI Sparv's dependency system will calculate a dependency graph and all the functions necessary for producing the desired output will be run automatically.

2.5 Reading and Writing Files

Sparv classes like `Annotation` and `Output` have built-in methods for reading and writing files (like `word.read()` and `out.write()` in the above example). A Sparv module should never read or write any files without using the provided class methods. This is to make sure that files are written to the correct places in the file structure so that they can be found by other modules. The read and write methods also make sure that Sparv's internal data format is handled correctly. Not using these provided methods can lead to procedures breaking if the internal data format or file structure is updated in the future.

2.6 Logging

Logging from Sparv modules is done with Python's logging library. Please use the provided `get_logger` wrapper when declaring your logger which takes care of importing the logging library and sets the correct module name in the log output:

```
from sparv.api import get_logger
logger = get_logger(__name__)
logger.error("An error was encountered!")
```

Any of the official Python logging levels may be used.

By default, Sparv will write log output with level WARNING and higher to the terminal. The user can change the log level with the flag `--log [LOGLEVEL]`. Most commands support this flag. The user can also choose to write the log output to a file by using the `--log-to-file [LOGLEVEL]` flag. The log file will receive the current date and timestamp as filename and can be found inside `logs/` in the corpus directory.

2.6.1 Progress bar

It is possible to add a progress bar for individual annotators by using the custom `progress()` logging method. To initialize the progress bar, call the `logger.progress()` method, either without an argument, or while supplying the total for the bar: `logger.progress(total=50)`. A progress bar initialized without a total will have to be provided with a total before it can be used. It is also possible to change the total later.

After the total has been set, call `progress()` again to update the progress. If not argument is supplied, the progress is advanced by 1. To advance by another amount, use the keyword argument `advance=`. To set the progress to a specific number, simply call the method with that number as the argument. See below for examples:

```
from sparv.api import get_logger
logger = get_logger(__name__)

# Initialize progress bar with no known total
logger.progress()

# Initialize bar with known total
logger.progress(total=50)

# Advance progress by 1
logger.progress()

# Advance progress by 2
logger.progress(advance=2)

# Set progress to 5
logger.progress(5)
```

2.7 Error Messages

When raising critical errors that should be displayed to the user (e.g. to tell the user that he/she did something wrong) you should use the SparvErrorMessage class. This will raise an exception (and thus stop the current Sparv process) and notify the user of errors in a friendly way without displaying the usual Python traceback.

```
from sparv.api import SparvErrorMessage

@annotator("Convert every word to uppercase")
def uppercase(word: Annotation = Annotation("<token:word>"),
              out: Output = Output("<token>:sbx_uppercase.upper"),
              important_config_variable: str = Config("sbx_uppercase.some_setting
                  ")):
    """Convert to uppercase."""
    # Make sure important_config_variable is set by the user
    if not important_config_variable:
        raise SparvErrorMessage("Please make sure to set the config variable '
            sbx_uppercase.some_setting'!")
    ...
```

2.8 Languages and varieties

It is possible to restrict the use of an annotator, exporter, installer or modelbuilder to one or more specific language(s). This is done by passing a list of ISO 639-3 language codes to the optional `language` parameter in the decorator:

```
@annotator("Convert every word to uppercase", language=["swe", "eng"])
def ...
```

Sparv functions are only available for use if one of their languages match the language in the corpus config file. If no language codes are provided in the decorator, the function is available for any corpus.

Sparv also supports language varieties which is useful when you want to write Sparv functions for a specific variety of a language. For instance, Sparv has some built-in annotators that are restricted to corpora with historical Swedish from the 1800's. They are marked with the language code `swe-1800`,

where `swe` is the ISO 639-3 code for Swedish and `1800` is an arbitrary string for this specific language variety. Sparv functions marked with `swe-1800` are available for corpora that are configured as follows:

```
metadata:
    language: "swe"
    variety: "1800"
```

Note that all functions marked with `swe` will also be available for these corpora.

2.9 Installing and Uninstalling Plugins

A Sparv plugin can be installed from the Python Package Index (PyPI), a remote public repository, or from a local directory stored anywhere on your machine. As long as the Sparv Pipeline is installed on your machine, you should be able to inject your plugin into the Sparv Pipeline code using pipx:

```
pipx inject sparv-pipeline [pointer-to-sparv-plugin]
```

So if you are trying to install the `sparv-sbx-uppercase` plugin and it exists on PyPI you can install it like this:

```
pipx inject sparv-pipeline sparv-sbx-uppercase
```

For installing it from a public repository from GitHub the install command looks something like this:

```
pipx inject sparv-pipeline https://github.com/spraakbanken/sparv-plugin-template/
    archive/main.zip
```

For installation from a local directory run this (from the directory containing your plugin):

```
pipx inject sparv-pipeline ./sparv-sbx-uppercase
```

After the injection the plugin functionality should be available, and the plugged-in module should be treated just like any other module within the Sparv Pipeline.

You can uninstall the plugin by running:

```
pipx runpip sparv-pipeline uninstall [name-of-sparv-plugin]
```

In this example `[name-of-sparv-plugin]` is `sparv-sbx-uppercase`.

2.10 Advanced Features

This section contains documentation for more advanced features which may be used but are not necessary for writing plugins.

2.10.1 Function Order

Sometimes one may want to create multiple Sparv functions that create the same output files (e.g. annotation files, export files or model files). In this case Sparv needs to be informed about the priority of these functions. Let's say that there are two functions `annotate()` and `annotate_backoff()` that both produce an annotation output called `mymodule.foo`. Ideally `mymodule.foo` should be produced by `annotate()` but if this function cannot be run for some reason (e.g. because it needs another annotation

file `mymodule.bar` that cannot be produced for some corpora), then you want `mymodule.foo` to be produced by `annotate_backoff()`. The priority of functions is stated with the `order` argument in the `@annotator`, `@exporter`, or `@modelbuilder` decorator. The integer value given by `order` will help Sparv decide which function to try to use first. A lower number indicates higher priority.

```
@annotator("Create foo annotation", order=1)
def annotate(
    out: Output = Output("mymodule.foo"),
    bar_input: Annotation = Annotation("mymodule.bar")):
    ...


@annotator("Create foo annotation when bar is not available", order=2)
def annotate_backoff(
    out: Output = Output("mymodule.foo")):
    ...
```

2.10.2 Preloaders

Preloader functions are used by the `sparv preload` command to speed up the annotation process. It works by preloading the Python module together with models or processes which would otherwise need to be loaded once for every source file.

A preload function is simply a function that takes a subset of the arguments from an annotator, and returns a value that is passed on to the annotator. Here is an example:

```
from sparv.api import Annotation, Model, Output, annotator


def preloader(model):
    """Preload POS-model."""
    return load_model(model)


@annotator("Part-of-speech tagging.",
           preloader=preloader,
           preloader_params=["model"],
           preloader_target="model_preloaded")
def pos_tag(word: Annotation = Annotation("<token:word>"),
            out: Output = Output("<token>:pos.tag"),
            model: Model = Model("pos.model"),
            model_preloaded=None):
    """Annotate tokens with POS tags."""
    if model_preloaded:
        model = model_preloaded
    else:
        model = load_model(model)
```

This annotator uses a model. It also has an extra argument called `model_preloaded` which can optionally take an already loaded model. In the decorator we point out the preloader function using the `preloader` parameter. `preloader_params` is a list of parameter names from the annotator, which the preloader needs as arguments. In this case it's only one: the `model` parameter. `preloader_target` points to a single parameter name of the annotator, which is the one that will recieve the return value from the preloader function.

When using the `sparv preload` command with this annotator, the preloader function will be run only once, and every time the annotator is used, it will get the preloaded model via the `model_preloaded` parameter.

The three decorator parameters `preloader`, `preloader_params` and `preloader_target` are all required when adding a preloader to an annotator. Additionally, there are two optional parameters that can be used: `preloader_shared` and `preloader_cleanup`.

`preloader_shared` is a boolean with a default value of True. By default, Sparv will run the preloader function only once, and if using `sparv preload` with multiple parallel processes, they will all share the result from the preloader. By setting `preloader_shared` to False, the preloader function will instead be run once per process. This is usually only needed when preloading processes, rather than models.

`preloader_cleanup` refers to a function, just like the `preloader` parameter. This function will be run after every (preloaded) use of the annotator. As arguments the cleanup function should take the same arguments as the preloader function, plus an extra argument for receiving the return value of the preloader function. It should return the same kind of object as the preloader function, which will then be used by Sparv as the new preloaded value. This is rarely needed, but one possible use case is when preloading a process that for some reason needs to be regularly restarted. The cleanup function would then keep track of when restarting is needed, call the preloader function to start a new process, and then return it.

3 Sparv Decorators

Sparv decorators are used to make functions known to the Sparv registry. Only decorated functions will automatically become part of the pipeline. When Sparv is run it will automatically search for decorated functions and scan their arguments, thus building an index of their inputs and outputs. This index is then used to build the dependency graph.

The available decorators are listed below. Every decorator (except for `@wizard`) has one mandatory argument, the description, which is a string describing what the function does. This is used for displaying help texts in the CLI. All other arguments are optional and default to `None`.

3.1 @annotator

A function decorated with `@annotator` usually takes some input (e.g. models, one or more arbitrary annotations like tokens, sentences, parts of speeches etc.) and outputs one or more new annotations.

Arguments:

- `name`: Optional name to use instead of the function name.
- `description`: Description of the annotator. Used for displaying help texts in the CLI.
- `config`: List of Config instances defining config options for the annotator.
- `language`: List of supported languages. If no list is supplied all languages are supported.
- `order`: If several annotators have the same output, this integer value will help decide which to try to use first. A lower number indicates higher priority.
- `wildcards`: List of wildcards used in the annotator function's arguments.
- `preloader`: Reference to a preloader function, used to preload models or processes.
- `preloader_params`: A list of names of parameters for the annotator, which will be used as arguments for the preloader.
- `preloader_target`: The name of the annotator parameter which should receive the return value of the preloader.
- `preloader_cleanup`: Reference to an optional cleanup function, which will be executed after each annotator use.
- `preloader_shared`: Set to False if the preloader result should not be shared among preloader processes.

Example:

```
@annotator("Part-of-speech tags and baseforms from TreeTagger",
           language=["bul", "est", "fin", "lat", "nld", "pol", "ron", "slk", "deu
               ", "eng", "fra", "spa", "ita", "rus"],
           config=[
               Config("treetagger.binary", "tree-tagger", description="TreeTagger
                    executable"),
               Config("treetagger.model", "treetagger/[metadata.language].par",
                   description="Path to TreeTagger model")
           ])
def annotate(lang: Language = Language(),
             model: Model = Model("[treetagger.model]"),
             tt_binary: Binary = Binary("[treetagger.binary]"),
             out_upos: Output = Output("<token>:treetagger.upos", cls="token:upos
                 ", description="Part-of-speeches in UD"),
             out_pos: Output = Output("<token>:treetagger.pos", cls="token:pos",
                 description="Part-of-speeches from TreeTagger"),
             out_baseform: Output = Output("<token>:treetagger.baseform",
                 description="Baseforms from TreeTagger"),
             word: Annotation = Annotation("<token:word>"),
             sentence: Annotation = Annotation("<sentence>")):
    ...
```

3.2 @importer

A function decorated with `@importer` is used for importing corpus files in a certain file format. Its job is to read a corpus file, extract the corpus text and existing markup (if applicable), and write annotation files for the corpus text and markup.

Importers do not use the `Output` class to specify its outputs. Instead, outputs may be listed using the `outputs` argument of the decorator. Any output that is to be used as explicit input by another part of the pipeline needs to be listed here, but additional unlisted outputs may also be created.

Two outputs are implicit (and thus not listed in `outputs`) but required for every importer: the corpus text, saved by using the `Text` class, and a list of the annotations created from existing markup, saved by using the `SourceStructure` class.

Arguments:

- `description`: Description of the importer. Used for displaying help texts in the CLI.
- `file_extension`: The file extension of the type of source this importer handles, e.g. "xml" or "txt".
- `name`: Optional name to use instead of the function name.
- `outputs`: A list of annotations and attributes that the importer is guaranteed to generate. May also be a Config instance referring to such a list. It may generate more outputs than listed, but only the annotations listed here will be available to use as input for annotator functions.
- `config`: List of Config instances defining config options for the importer.

Example:

```
@importer("TXT import", file_extension="txt", outputs=["text"])
def parse(source_file: SourceFilename = SourceFilename(),
          source_dir: Source = Source(),
          prefix: str = "",
          encoding: str = util.constants.UTF8,
          normalize: str = "NFC") -> None:
    ...
```

3.3 @exporter

A function decorated with `@exporter` is used to produce "final" output (also called export), typically combining information from multiple annotations into one file. Output produced by an exporter is usually not used as input in any another module. An export can consist of one file per input corpus file or it can combine information from all input files into one output file.

Arguments:

- `description`: Description of the exporter. Used for displaying help texts in the CLI.
- `name`: Optional name to use instead of the function name.
- `config`: List of Config instances defining config options for the exporter.
- `language`: List of supported languages. If no list is supplied all languages are supported.
- `order`: If several exporters have the same output, this integer value will help decide which to try to use first. A lower number indicates higher priority.
- `abstract`: Set to True if this exporter has no output.

Example:

```
@exporter("Corpus word frequency list (withouth Swedish annotations)", order=2,
    config=[
    Config("stats_export.delimiter", default="\t", description="Delimiter
        separating columns"),
    Config("stats_export.cutoff", default=1, description="The minimum frequency a
         word must have in order to be included in the result")
])
def freq_list_simple(corpus: Corpus = Corpus(),
                     source_files: AllSourceFilenames = AllSourceFilenames(),
                     word: AnnotationAllSourceFiles = AnnotationAllSourceFiles("<
                         token:word>"),
                     pos: AnnotationAllSourceFiles = AnnotationAllSourceFiles("<
                         token:pos>"),
                     baseform: AnnotationAllSourceFiles =
                         AnnotationAllSourceFiles("<token:baseform>"),
                     out: Export = Export("stats_export.frequency_list/stats_[
                         metadata.id].csv"),
                     delimiter: str = Config("stats_export.delimiter"),
                     cutoff: int = Config("stats_export.cutoff")):
    ...
```

3.4 @installer

A function decorated with `@installer` is used to copy a corpus export to a remote server.

Arguments:

- `description`: Description of the installer. Used for displaying help texts in the CLI.
- `name`: Optional name to use instead of the function name.
- `config`: List of Config instances defining config options for the installer.
- `language`: List of supported languages. If no list is supplied all languages are supported.

Example:

```
@installer("Copy compressed XML to remote host", config=[
    Config("xml_export.export_host", "", description="Remote host to copy XML
        export to."),
    Config("xml_export.export_path", "", description="Path on remote host to copy
         XML export to.")
])
def install(corpus: Corpus = Corpus(),
            xmlfile: ExportInput = ExportInput("xml_export.combined/[metadata.id
                ].xml.bz2"),
            out: OutputCommonData = OutputCommonData("xml_export.
                install_export_pretty_marker"),
            export_path: str = Config("xml_export.export_path"),
            host: str = Config("xml_export.export_host")):
    ...
```

3.5 @modelbuilder

A function decorated with `@modelbuilder` is used to setup a model used by other Sparv components (typically annotators). Setting up a model could for example mean downloading a file, unzipping it, converting it into a different format and saving it in Sparv's data directory. A model is usually not specific to one corpus. Once a model is setup on your system it will be available for any corpus.

Arguments:

- `description`: Description of the installer. Used for displaying help texts in the CLI.
- `name`: Optional name to use instead of the function name.
- `config`: List of Config instances defining config options for the installer.
- `language`: List of supported languages. If no list is supplied all languages are supported.
- `order`: If several modelbuilders have the same output, this integer value will help decide which to try to use first. A lower number indicates higher priority.

Example:

```
@modelbuilder("Sentiment model (SenSALDO)", language=["swe"])
def build_model(out: ModelOutput = ModelOutput("sensaldo/sensaldo.pickle")):
    ...
```

3.6 @wizard

A function decorated with `@wizard` is used to generate questions for the corpus config wizard.

Arguments:

- `config_keys`: a list of config keys to be set or changed by this function.
- `source_structure`: Set to `True` if the function needs access to a `SourceStructureParser` instance (the one holding information on the structure of the source files. Default: `False`

Example:

```
@wizard(["export.source_annotations"], source_structure=True)
def import_wizard(answers, structure: SourceStructureParser):
    ...
```

4 Sparv Classes

Sparv classes are used to represent common data types within Sparv such as annotations and models. They are used for type hints and default arguments within the signatures of decorated functions. Sparv uses these classes to figure out what inputs and outputs a function has, which is essential for building dependencies between annotations. Sparv classes also provide useful methods such as methods for reading and writing annotations. Using these, an annotator can read and write annotation files without the need to know anything about Sparv's internal data format. Below is a list with all the available Sparv classes, their arguments, properties, and public methods.

4.1 AllSourceFilenames

An instance of this class holds a list with the names of all source files. It is typically used by exporter functions that combine annotations from all source files.

4.2 Annotation

An instance of this class represents a regular annotation tied to one source file. This class is used when an annotation is needed as input for a function, e.g. `Annotation("<token:word>")`.

Arguments:

- `name`: The name of the annotation.
- `source_file`: The name of the source file.
- `is_input`: If set to `False` the annotation won't be added to the rule's input. Default: `True`

Properties:

- `has_attribute`: Return True if the annotation has an attribute.
- `annotation_name`: Get annotation name (excluding name of any attribute).
- `attribute_name`: Get attribute name (excluding name of annotation).

Methods:

- `split()`: Split name into annotation name and attribute.
- `exists()`: Return True if annotation file exists.
- `read(allow_newlines: bool = False)`: Yield each line from the annotation.
- `get_children(child: BaseAnnotation, orphan_alert=False, preserve_parent_annotation_order=False)`: Return two lists. The first one is a list with n (= total number of parents) elements where every element is a list of indices in the child annotation. The second one is a list of orphans, i.e. containing indices in the child annotation that have no parent. Both parents and children are sorted according to their position in the source file, unless preserve_parent_annotation_order is set to True, in which case the parents keep the order from the parent annotation.
- `get_parents(parent: BaseAnnotation, orphan_alert: bool = False)`: Return a list with n (= total number of children) elements where every element is an index in the parent annotation. Return None when no parent is found.
- `read_parents_and_children(parent, child)`: Read parent and child annotations. Reorder them according to span position, but keep original index information.
- `read_attributes(annotations: Union[List[BaseAnnotation], Tuple[BaseAnnotation, ...]], with_annotation_name: bool = False, allow_newlines : bool = False)`: Yield tuples of multiple attributes on the same annotation.
- `read_spans(decimals=False, with_annotation_name=False)`: Yield the spans of the annotation.
- `create_empty_attribute()`: Return a list filled with None of the same size as this annotation.
- `get_size()`: Get the number of values.

4.3 AnnotationAllSourceFiles

Regular annotation but the source filename must be specified for all actions. Use as input to an annotator function to require the specificed annotation for every source file in the corpus.

Arguments:

- `name`: The name of the annotation.

Properties:

- `annotation_name`: Get annotation name (excluding name of any attribute).
- `attribute_name`: Get attribute name (excluding name of annotation)

Methods:

- `split()`: Split name into annotation name and attribute.
- `read(source_file: str)`: Yield each line from the annotation.
- `read_spans(source_file: str, decimals=False, with_annotation_name=False)`: Yield the spans of the annotation.
- `create_empty_attribute(source_file: str)`: Return a list filled with None of the same size as this annotation.
- `exists(source_file: str)`: Return True if annotation file exists.
- `get_size(source_file: str)`: Get the number of values.

4.4 AnnotationCommonData

Like `AnnotationData`, an instance of this class represents an annotation with arbitrary data, but `AnnotationCommonData` is used for data that applies to the whole corpus (i.e. data that is not specific to one source file).

Arguments:

- `name`: The name of the annotation.

Properties:

- `annotation_name`: Get annotation name (excluding name of any attribute).
- `attribute_name`: Get attribute name (excluding name of annotation)

Methods:

- `split()`: Split name into annotation name and attribute.
- `read()`: Read arbitrary corpus level string data from annotation file.

4.5 AnnotationData

This class represents an annotation holding arbitrary data, i.e. data that is not tied to spans in the corpus text.

Arguments:

- `name`: The name of the annotation.
- `source_file`: The name of the source file.

Properties:

- `has_attribute`: Return True if the annotation has an attribute.
- `annotation_name`: Get annotation name (excluding name of any attribute).
- `attribute_name`: Get attribute name (excluding name of annotation).

Methods:

- `split()`: Split name into annotation name and attribute.
- `exists()`: Return True if annotation file exists.
- `read(source_file: Optional[str] = None)`: Read arbitrary string data from annotation file.

4.6 AnnotationDataAllSourceFiles

Like `AnnotationData`, this class is used for annotations holding arbitrary data but the source file must be specified for all actions. Use as input to an annotator to require the specificed annotation for every source file in the corpus.

Arguments:

- `name`: The name of the annotation.

Properties:

- `annotation_name`: Get annotation name (excluding name of any attribute).
- `attribute_name`: Get attribute name (excluding name of annotation)

Methods:

- `split()`: Split name into annotation name and attribute.
- `exists()`: Return True if annotation file exists.
- `read(source_file: Optional[str] = None)`: Read arbitrary string data from annotation file.

4.7 Binary

An instance of this class holds a path to a binary executable. This may be a path relative to the `bin` path inside the Sparv data directory. This class is often used to define a prerequisite for an annotator function.

Arguments:

- Path to binary executable.

4.8 BinaryDir

An instance of this class holds the path to a directory containing executable binaries. This may be a path relative to the `bin` path inside the Sparv data directory.

Arguments:

- Path to directory containing executable binaries.

4.9 Config

An instance of this class holds a configuration key name and its default value.

Arguments:

- `name`: The name of the configuration key.
- `default`: An optional default value of the configuration key.
- `description`: An obligatory description.

4.10 Corpus

An instance of this class holds the name (ID) of the corpus.

4.11 SourceFilename

An instance of this class holds the name of a source file.

4.12 Export

An instance of this class represents an export file. This class is used to define an output of an exporter function.

Arguments:

- The export directory and filename pattern (e.g. `"xml_export.pretty/[xml_export.filename]"`). The export directory must contain the module name as a prefix, or be equal to the module name.

4.13 ExportAnnotations

List of annotations to be included in the export. This list is defined in the corpus configuration. Annotation files for the current source file will automatically be added as dependencies when using this class, unless `is_input` is set to `False`.

Arguments:

- `config_name`: The config variable pointing out what annotations to include.
- `is_input`: If set to `False` the annotations won't be added to the rule's input. Default: `True`

4.14 ExportAnnotationsAllSourceFiles

List of annotations to be included in the export. This list is defined in the corpus configuration. Annotation files for *all* source files will automatically be added as dependencies when using this class, unless `is_input` is set to `False`. With `is_input` set to `False`, there is no difference between using `ExportAnnotationsAllSourceFiles` and `ExportAnnotations`.

Arguments:

- `config_name`: The config variable pointing out what annotations to include.
- `is_input`: If set to `False` the annotations won't be added to the rule's input. Default: `True`

4.15 ExportInput

Export directory and filename pattern, used as input. Use this class if you need export files as input in another function.

Arguments:

- `val`: The export directory and filename pattern (e.g. `"xml_export.pretty/[xml_export.filename]"`).
- `all_files`: Set to `True` to get the export for all source files. Default: `False`

4.16 Headers

List of header annotation names for a given source file.

Arguments:

- The name of the source file.

Methods:

- `read()`: Read the headers file and return a list of header annotation names.
- `write(header_annotations: List[str])`: Write headers file.
- `exists()`: Return True if headers file exists for this source file.

4.17 Language

In instance of this class holds information about the luanguage of the corpus. This information is retrieved from the corpus configuration and is specified as ISO 639-3 code.

4.18 Model

An instance of this class holds a path to a model file relative to the Sparv model directory. This class is typically used as input to annotator functions.

Arguments: - `name`: The path to the model file relative to the model directory.

Properties:: - `path`: The path to the model file as a `pathlib.Path` object.

Methods:: - `write(data)`: Write arbitrary string data to models directory. - `read()`: Read arbitrary string data from file in models directory. - `write_pickle(data, protocol=-1)`: Dump `data` to pickle file in models directory. - `read_pickle()`: Read pickled data from file in models directory. - `download(url: str)`: Download file from `url` and save to modeldir. - `unzip()`: Unzip zip file inside modeldir. - `ungzip(out: str)`: Unzip gzip file inside modeldir. - `remove(raise_errors: bool = False)`: Remove model file from disk. If `raise_errors` is set to `True` an error will be raised if the file cannot be removed (e.g. if it does not exist).

4.19 ModelOutput

This class is very similar to `Model` but it is used as output of a modelbuilder.

Arguments: - `name`: The name of the annotation. - `description`: An optional description.

Properties:: - `path`: The path to the model file as a `pathlib.Path` object.

Methods:: - `write(data)`: Write arbitrary string data to models directory. - `read()`: Read arbitrary string data from file in models directory. - `write_pickle(data, protocol=-1)`: Dump `data` to pickle file in models directory. - `read_pickle()`: Read pickled data from file in models directory. - `download(url: str)`: Download file from `url` and save to modeldir. - `unzip()`: Unzip zip file inside modeldir. - `ungzip(out: str)`: Unzip gzip file inside modeldir. - `remove(raise_errors: bool = False)`: Remove model file from disk. If `raise_errors` is set to `True` an error will be raised if the file cannot be removed (e.g. if it does not exist).

4.20 Output

Regular annotation or attribute used as output (e.g. of an annotator function).

Arguments: - `name`: The name of the annotation. - `cls`: The annotation class of the output. - `description`: An optional description. - `source_file`: The name of the source file.

Methods:

- `split()`: Split name into annotation name and attribute.
- `write(values, append: bool = False, allow_newlines: bool = False, source_file: Optional[str] = None)`: Write an annotation to file. Existing annotation will be overwritten. 'values' should be a list of values.
- `exists()`: Return True if annotation file exists.

4.21 OutputAllSourceFiles

Similar to `Output` this class represents a regular annotation or attribute used as output, but the source file must be specified for all actions.

Arguments: - `name`: The name of the annotation. - `cls`: The annotation class of the output. - `description`: An optional description.

Methods:

- `split()`: Split name into annotation name and attribute.
- `write(values, source_file: str, append: bool = False, allow_newlines: bool = False)`: Write an annotation to file. Existing annotation will be overwritten. 'values' should be a list of values.
- `exists(source_file: str)`: Return True if annotation file exists.

4.22 OutputCommonData

Similar to `OutputData` but for a data annotation that is valid for the whole corpus.

Arguments: - `name`: The name of the annotation. - `cls`: The annotation class of the output. - `description`: An optional description.

Methods:

- `split()`: Split name into annotation name and attribute.
- `write(value, append: bool = False)`: Write arbitrary corpus level string data to annotation file.

4.23 OutputData

This class represents an annotation holding arbitrary data (i.e. data that is not tied to spans in the corpus text) that is used as output.

Arguments: - `name`: The name of the annotation. - `cls`: The annotation class of the output. - `description`: An optional description. - `source_file`: The name of the source file.

Methods:

- `split()`: Split name into annotation name and attribute.
- `write(value, append: bool = False)`: Write arbitrary corpus level string data to annotation file.
- `exists()`: Return True if annotation file exists.

4.24 OutputDataAllSourceFiles

Like `OutputData`, this class is used for annotations holding arbitrary data and that is used as output, but the source file must be specified for all actions.

Arguments: - `name`: The name of the annotation. - `cls`: The annotation class of the output. - `description`: An optional description.

Methods:

- `split()`: Split name into annotation name and attribute.
- `write(value, source_file: str, append: bool = False)`: Write arbitrary corpus level string data to annotation file.
- `exists(source_file: str)`: Return True if annotation file exists.

4.25 Source

An instance of this class holds a path to the directory containing input files.

Arguments:

- Path to directory containing input files.

Methods:

- `get_path(source_file: SourceFilename, extension: str)`: Get path to a specific source file.

4.26 SourceAnnotations

List of source annotations to be included in the export. This list is defined in the corpus configuration.

Arguments:

- `config_name`: The config variable pointing out what source annotations to include.

4.27 SourceAnnotationsAllSourceFiles

List of source annotations to be included in the export. This list is defined in the corpus configuration. This differs from `SourceAnnotations` in that the source annotations structure file (created by using `SourceStructure`) of *every* source file will be added as dependencies.

Arguments:

- `config_name`: The config variable pointing out what source annotations to include.

4.28 SourceStructure

Every annotation name available in a source file.

Arguments:

- The name of the source file.

Methods:

- `read()`: Read structure file.
- `write(structure)`: Sort the source file's annotation names and write to structure file.

4.29 SourceStructureParser

This is an abstract class that should be implemented by an importer's structure parser.

Arguments:

- `source_dir: pathlib.Path`: Path to corpus source files.

Methods:

- `setup()`: Return a list of wizard dictionaries with questions needed for setting up the class. Answers to the questions will automatically be saved to self.answers.

4.30 Text

An instance of this class represents the corpus text.

Arguments:

- `source_file`: The name of the source file.

Methods:

- `read()`: Get corpus text.
- `write(text)`: Write text to the designated file of a corpus. `text` is a unicode string.

4.31 Wildcard

An instance of this class holds wildcard information. It is typically used in the `wildcards` list passed as an argument to the `@annotator` decorator, e.g.:

```
@annotator("Number {annotation} by relative position within {parent}", wildcards
    =[
    Wildcard("annotation", Wildcard.ANNOTATION),
    Wildcard("parent", Wildcard.ANNOTATION)
])
```

Arguments:

- `name`: The name of the wildcard.

- `type`: The type of the wildcard. One of `Wildcard.ANNOTATION`, `Wildcard.ATTRIBUTE`, `Wildcard.ANNOTATION_ATTRIBUTE`, `Wildcard.OTHER`. Defaults to `Wildcard.OTHER`.
- `description`: An optional description.

5 Config Parameters

A Sparv user can steer and customise the use of Sparv functions to some extent by setting config parameters in the corpus config file. Each function decorated with a Sparv decorator (except for wizard functions) may take a `config` argument which contains a list of config parameters, their descriptions and optional default values. The config parameters declared here can then be referenced in the function's arguments:

```
@annotator("Dependency parsing using MaltParser", language=["swe"], config=[
    Config("malt.jar", default="maltparser-1.7.2/maltparser-1.7.2.jar",
           description="Path name of the executable .jar file"),
    Config("malt.model", default="malt/swemalt-1.7.2.mco", description="Path to
        Malt model")
])
def annotate(maltjar: Binary = Binary("[malt.jar]"),
             model: Model = Model("[malt.model]"),
             ...):
    ...
    process = maltstart(maltjar, model)
    ...
```

Config parameters can also be declared in the module's `__init__.py` file, using the global variable `__config__`:

```
__config__ = [
    Config("korp.remote_host", description="Remote host to install to"),
    Config("korp.mysql_dbname", description="Name of database where Korp data
        will be stored")
]
```

A Sparv function must never try to read any config values inside the function body. Config parameters are always accessed via the function's arguments as shown in the above example. It is important to let the Sparv core handle the reading of the corpus configuration in order for the internal config hierarchy and config inheritance to be respected and treated correctly.

To be able to use a config parameter inside a Sparv function it must first be declared in a Sparv decorator or in a module's init file. However, a config parameter used inside a Sparv function does not necessarily have to be declared in the decorator belonging to that same function, but the declaration may be done in a decorator belonging to a different Sparv function, or even a different module.

Please note that it is mandatory to set a description for each declared config parameter. These descriptions are displayed to the user when lising modules with the `sparv modules` command.

5.1 Config hierarchy

When Sparv processes the corpus configuration it will look for config values in four different places in the indicated priority order: 1. the corpus configuration file 2. a parent corpus configuration file 2. the default configuration file in the Sparv data directory 3. config default values defined in the Sparv decorators (as shown above)

This means that if a config parameter is given a default value in a Sparv decorator it can be overridden by the default configuration file which in turn can be overridden by the user's corpus config file.

5.2 Config Inheritance

Sparv importers and exporters inherit their configuration from the more general config categories `import` and `export`. For example when setting `export.annotations` as follows:

```
export:
    annotations:
        - <token>:hunpos.pos
        - <token>:saldo.baseform
```

the config parameter `csv_export.annotations` belonging to the CSV exporter will automatically be set to the same value (unless it is explicitly set to another value in the corpus config file):

```
csv_export:
    annotations:
        - <token>:hunpos.pos
        - <token>:saldo.baseform
```

This means that when writing an importer or exporter you should try to use the same predefined configuration key names wherever it makes sense unless you have a good reason not to. Here is a list of all the existing configuration keys for the `import` and the `export` categories that are inherited by importers and exporters:

Inheritable configuration keys for `import`:

config key	description
`text_annotation`	The annotation representing one text. Any text-level annotations will be attached to this annotation.
`encoding`	Encoding of source file. Defaults to UTF-8.
`keep_control_chars`	Set to True if control characters should not be removed from the text.
`normalize`	Normalize input using any of the following forms: 'NFC', 'NFKC', 'NFD', and 'NFKD'.
`source_dir`	The path to the directory containing the source files relative to the corpus directory.

Inheritable configuration keys for `export`:

config key	description
`default`	Exports to create by default when running 'sparv run'.
`source_annotations`	List of annotations from the source file to be kept.
`annotations`	List of automatic annotations to include.
`word`	The token strings to be included in the export.
`remove_module_namespaces`	Set to false if module name spaces should be kept in the export.
`sparv_namespace`	A string representing the name space to be added to all annotations created by Sparv.
`source_namespace`	A string representing the name space to be added to all annotations present in the source.
`scramble_on`	Chunk to scramble the XML export on.

6 Wildcards

Some annotators use wildcards in their input and output. This is a useful mechanism that makes it possible for an annotator to produce many different annotations with different wildcard values. The annotator `misc.number_by_position` is an example of such an annotator. Its output is defined as `Output("{annotation}:misc.number_position")`. The wildcard `{annotation}` can be replaced with any annotation, and the annotator will produce a new attribute belonging to the spans of the annotation. So if a user asks for the annotation `<sentence>:misc.number_position` (by including it in one of the export lists in the corpus config) Sparv will add numbers to every sentence, when asking for `document:misc.number_position` Sparv will add a number attribute to the annotation called `document` and so on.

In a way wildcards are similar to config variables as they add some level of customization to annotators. The main difference is that a config variable is set explicitly in the corpus configuration while a wildcard receives its value automatically when referenced in an annotation.

In the function arguments wildcards are always marked with curly brackets, and they must be declared in the wildcard argument of the `@annotator` decorator as in the following example:

```
@annotator("Number {annotation} by position", wildcards=[Wildcard("annotation",
    Wildcard.ANNOTATION)])
def number_by_position(out: Output = Output("{annotation}:misc.number_position"),
                       chunk: Annotation = Annotation("{annotation}"),
                       ...):
    ...
```

It probably goes without saying that in order for a wildcard to make sense, the same wildcard variable must be used in the name of an input annotation (typically `Annotation`) and the name of an output annotation (e.g. `Output`) in the same annotation function.

An annotator may also have multiple wildcards as shown in the following example:

```
@annotator("Number {annotation} by relative position within {parent}", wildcards
    =[
    Wildcard("annotation", Wildcard.ANNOTATION),
    Wildcard("parent", Wildcard.ANNOTATION)
])
def number_relative(out: Output = Output("{annotation}:misc.number_rel_{parent}")
    ,
                    parent: Annotation = Annotation("{parent}"),
                    child: Annotation = Annotation("{annotation}"),
                    ...):
    ...
```

7 Utilities

Sparv has a number of utility functions, classes and constants that are not specific to any particular module. Most of them are imported from `sparv.api.util` and its submodules, e.g.:

```
from sparv.api.util.system import call_binary
```

7.1 Constants

`sparv.api.util.constants` contains the following constants:

- `DELIM = "|"` Delimiter char to put between ambiguous results
- `AFFIX = "|"` Character to put before and after results to mark a set
- `SCORESEP = ":"` Character that separates an annotation from a score
- `COMPSEP = "+"` Character to separate compound parts
- `UNDEF = "__UNDEF__"` Value for undefined annotations
- `OVERLAP_ATTR = "overlap"` Name for automatically created overlap attributes
- `SPARV_DEFAULT_NAMESPACE = "sparv"` Namespace to be used in case annotation names collide and `sparv_namespace` is not set in config
- `UTF8 = "UTF-8"` UTF-8 encoding
- `LATIN1 = "ISO-8859-1"` Latin-1 encoding
- `HEADER_CONTENTS = "contents"` Name of annotation containing header contents

7.2 Export Utils

`sparv.api.util.export` provides util functions used for preparing data for export.

7.2.1 gather_annotations()

Calculate the span hierarchy and the annotation_dict containing all annotation elements and attributes. Returns a `spans_dict` and an `annotation_dict` if `flatten` is set to `True`, otherwise `span_positions` and `annotation_dict`.

Arguments:

- `annotations`: A list of annotations to include.
- `export_names`: Dictionary that maps from annotation names to export names.
- `header_annotations`: A list of header annotations.
- `source_file`: The source filename.
- `flatten`: Whether to return the spans as a flat list. Default: `True`
- `split_overlaps`: Whether to split up overlapping spans. Default: `False`

7.2.2 get_annotation_names()

Get a list of annotations, token attributes and a dictionary with translations from annotation names to export names.

Arguments:

- `annotations`: List of elements:attributes (annotations) to include.
- `source_annotations`: List of elements:attributes from the source file to include. If not specified, everything will be included.
- `source_file`: Name of the source file.
- `source_files`: List of names of source files (alternative to `source_file`).
- `token_name`: Name of the token annotation.
- `remove_namespaces`: Remove all namespaces in export_names unless names are ambiguous. Default: `False`

- `keep_struct_names`: For structural attributes (anything other than token), include the annotation base name (everything before ":") in export_names (used in cwb encode). Default: `False`
- `sparv_namespace`: The namespace to be added to all Sparv annotations.
- `source_namespace`: The namespace to be added to all annotations present in the source.

7.2.3 get_header_names()

Get a list of header annotations and a dictionary for renamed annotations.

Arguments:

- `header_annotation_names`: List of header elements:attributes from the source file to include. If not specified, everything will be included.
- `source_file`: Name of the source file.
- `source_files`: List of names of source files (alternative to `source_file`).

7.2.4 scramble_spans()

Reorder chunks according to `chunk_order` and open/close tags in the correct order.

Arguments:

- `span_positions`: The original span positions (usually retrieved from `gather_annotations()`).
- `chunk_name`: The name of the annotation to scramble on.
- `chunk_order`: Annotation containing the new order of the chunk.

7.3 Install Utils

`sparv.api.util.install` provides util functions used for installing corpora onto remote locations.

7.3.1 install_directory()

Rsync every file from a local directory to a target host. The target path is extracted from filenames by replacing "#" with "/".

Arguments:

- `host`: The remote host to install to.
- `directory`: The directory to sync.

7.3.2 install_file()

Rsync a file to a target host.

Arguments:

- `local_file`: Path to the local file to sync.
- `host`: The remote host to install to.
- `remote_file`: The name of the resulting file on the remote host.

7.3.3 install_mysql()

Insert tables and data from local SQL-file to remote MySQL database.

Arguments:

- `host`: The remote host to install to.
- `db_name`: Name of the remote database.
- `sqlfile`: Path to a local SQL file, or multiple paths separated by whitespaces.

7.3.4 install_mysql_dump()

Copy selected tables (including data) from local to remote MySQL database.

Arguments:

- `host`: The remote host to install to.
- `db_name`: Name of the remote database.
- `tables`: Names of SQL tables to be copied separated by whitespaces.

7.4 System Utils

`sparv.api.util.system` provides functions related to starting and stopping processes, creating directories etc.

7.4.1 call_binary()

Call a binary with `arguments` and `stdin` and return a pair `(stdout, stderr)`.

Arguments:

- `name`: Name of the binary call.
- `arguments`: Arguments to pass to the call. Defaults to `()`.
- `stdin`: Stdin input to pass to the call. Defaults to `""`.
- `raw_command`: Don't use this unless you really have to! String holding the raw command that will be executed through the shell. Defaults to `None`.
- `search_paths`: List of paths where to look for the binary `name`, in addition to the environment variable PATH. Defaults to `()`.
- `encoding`: Encoding to use for `stdin`. Defaults to `None`.
- `verbose`: If set to `True` pipes all stderr output from the subprocess to stderr in the terminal, and an empty string is returned as the stderr component. Defaults to `False`.
- `use_shell`: Don't use this unless you really have to! If set to `True` the binary will be executed through the shell. Defaults to `False`. Is automatically set to `True` when using `raw_command`.
- `allow_error`: If set to `False` an exeption is raised if stderr is not empty and stderr and stdout will be logged. Defaults to `False`.
- `return_command`: If set to `True` the process is returned. Defaults to `False`.

7.4.2 call_java()

Call Java with a jar file, command line arguments and stdin. Returns a pair `(stdout, stderr)`.

Arguments:

- `jar`: The name of the jar file to call.
- `arguments`: Arguments to pass to the call. Defaults to `()`.
- `options`: List of Java options to pass to the call. Defaults to `[]`,
- `stdin`: Stdin input to pass to the call. Defaults to `""`.
- `search_paths`: List of paths where to look for the binary `name`, in addition to the environment variable PATH. Defaults to `()`.
- `encoding`: Encoding to use for `stdin`. Defaults to `None`.
- `verbose`: If set to `True` pipes all stderr output from the subprocess to stderr in the terminal, and an empty string is returned as the stderr component. Defaults to `False`.
- `return_command`: If set to `True` the process is returned. Defaults to `False`.

7.4.3 clear_directory()

Create a new empty directory. Remove it's contents if it already exists.

Arguments:

- `path`: Path to the directory to be created.

7.4.4 find_binary()

Search for the binary for a program. Returns the path to binary, or `None` if not found.

Arguments:

- `name`: Name of the binary, either a string or a list of strings with alternative names.
- `search_paths`: List of paths where to look, in addition to the environment variable PATH.
- `executable`: Set to `False` to not fail when binary is not executable. Defaults to `True`.
- `allow_dir`: Set to `True` to allow the target to be a directory instead of a file. Defaults to `False`.
- `raise_error`: If set to `True` raises error if binary could not be found. Defaults to `False`.

7.4.5 kill_process()

Kill a process, and ignore the error if it is already dead.

Arguments:

- `process`: The process to be killed.

7.4.6 rsync()

Transfer files and/or directories using rsync. When syncing directories, extraneous files in destination dirs are deleted.

Arguments:

- `local`: Path to a local file or directory.
- `host`: The remote host to rsync to.
- `remote`: Path on the remote host to rsync to. Defaults to `None`. If not provided, the path will be the same as on the local machine.

7.5 Tagsets

`sparv.api.util.tagsets` is a subpackage with modules containing functions and objects related to tagset conversions.

7.5.1 tagmappings.join_tag()

Convert a complex SUC or SALDO tag record into a string.

Arguments:

- `tag`: The tag to convert to a string. Can be a dict (`{'pos': pos, 'msd': msd}`) or a tuple (`(pos, msd)`)
- `sep`: The separator to be used. Default: "."

7.5.2 tagmappings.mappings

Dictionary containing mappings (dictionaries) for of part-of-speech tag mappings between different tag sets.

7.5.3 pos_to_upos()

Map POS tags to Universal Depenendy POS tags. This only works if there is a conversion function in `util.tagsets.pos_to_upos` for the given language and tagset.

Arguments:

- `pos`: The part-of-speech tag to convert.
- `lang`: The language code.
- `tagset`: The name of the tagset that `pos` belongs to.

7.5.4 tagmappings.split_tag()

Split a SUC or Saldo tag string ('X.Y.Z') into a tuple ('X', 'Y.Z') where 'X' is a part of speech and 'Y', 'Z' etc. are morphologic features (i.e. MSD tags).

Arguments:

- `tag`: The tag string to convert into a tuple.
- `sep`: The separator to split on. Default: "."

7.5.5 suc_to_feats()

Convert SUC MSD tags into UCoNNL feature list (universal morphological features). Returns a list of universal features.

Arguments:

- `pos`: The SUC part-of-speech tag.
- `msd`: The SUC MSD tag.
- `delim`: The delimiter separating the features in `msd`. Default: "."

7.5.6 tagmappings.tags

Dictionary containing sets of part-of-speech tags.

7.6 Miscellaneous Utils

`sparv.api.util.misc` provides miscellaneous util functions.

7.6.1 cwbset()

Take an iterable object and return a set in the format used by Corpus Workbench.

Arguments:

- `values`: An iterable containing some string values.
- `delimiter`: Character that delimits the elements in the resulting set. Default: "|"
- `affix`: Character that the resulting set starts and ends with. that Default: "|"
- `sort: Set to` True `if you want to values to be sorted. Default:` False'
- `maxlength`: Maximum length in characters for the resulting set. Default: 4095
- `encoding`: Encoding of `values`. Default: "UTF-8"

7.6.2 indent_xml()

Add pretty-print indentation to an XML tree.

Arguments:

- `elem`: The XML tree (`xml.etree.ElementTree.Element`) to indent.
- `level`: The indentation start level. Default: `0`
- `indentation`: The indentation to add on each level. Default: ' '

7.6.3 parse_annotation_list()

Take a list of annotation names and possible export names, and return a list of tuples. Each list item will be split into a tuple by the string ' as '. Each tuple will contain 2 elements. If there is no' as ' in the string, the second element will be None.

Arguments:

- `annotation_names`: List of annotations.

- `all_annotations`: List of annotations. If there is an element called '...' everything from all_annotations will be included in the result, except for the elements that are prefixed with 'not'. Default: `[]`
- `add_plain_annotations`: Plain annotations (without attributes) will be added if needed, unless add_plain_annotations is set to False. Make sure to disable add_plain_annotations if the annotation names may include classes or config variables. Default: `True`

7.6.4 PickledLexicon

Class for reading basic pickled lexicon and looking up keys.

Arguments:

- `picklefile`: A `pathlib.Path` or `Model` object pointing to a pickled lexicon.
- `verbose`: Logs status updates upon reading the lexicon if set to `True`. Default: `True`

Methods:

- `lookup(key, default=set())`: Look up `key` in the lexicon. Return `default` if `key` is not found.

7.6.5 remove_control_characters()

Remove control characters from `text`, except for those in `keep`.

Arguments:

- `text`: String to remove control characters from.
- `keep`: List of control characters to keep. Default: `["\n", "\t", "\r"]`

7.6.6 remove_formatting_characters()

Remove formatting characters from `text`, except for those in `keep`.

Arguments:

- `text`: String to remove formatting characters from.
- `keep`: List of formatting characters to keep. Default: `[]`

7.6.7 set_to_list()

Turn a set string into a list.

Arguments:

- `setstring`: A string that can be converted into a list by stripping it of `affix` and splitting the elements on `delimiter`.
- `delimiter`: Character that delimits the elements in `setstring`. Default: "|"
- `affix`: Character that `setstring` starts and ends with. that Default: "|"

7.6.8 test_lexicon()

Test the validity of a lexicon. Takes a dictionary (lexicon) and a list of test words that are expected to occur as keys in the lexicon. Prints the value for each test word.

Arguments:

- `lexicon`: A dictionary.
- `testwords`: An iterable containing strings that are expected to occur as keys in `lexicon`.

7.7 Error Messages and Logging

The `SparvErrorMessage` exception and `get_logger` function are integral parts of the Sparv pipeline, and unlike other utilities on this page, they are found directly under `sparv.api`.

7.7.1 SparvErrorMessage

Exception (class) used to notify users of errors in a friendly way without displaying traceback. Its usage is described in the Writing Sparv Plugins section.

> **Note:** Only the `message` argument should be used when raisning this exception in a Sparv module.

Arguments:

- `message`: The error message to display.
- `module`: Name of the module where the error occurred (optional, not used in Sparv modules). Default: ""
- `function`: Name of the function where the error occurred (optional, not used in Sparv modules). Default: ""

7.7.2 get_logger()

Get a logger that is a child of `sparv.modules`. Its usage is described in the Writing Sparv Plugins section.

Arguments:

- `name`: The name of the current module (usually `__name__`)

GU-ISS, Forskningsrapporter från Institutionen för svenska, flerspråkighet och språkteknologi, är en oregelbundet utkommande serie, som i enkel form möjliggör spridning av institutionens skriftliga produktion. Det främsta syftet med serien är att fungera som en kanal för preliminära texter som kan bearbetas vidare för en slutgiltig publicering. Varje enskild författare ansvarar för sitt bidrag.

GU-ISS, Research reports from the Department of Swedish, Multilingualism, Language Technology is an irregular report series intended as a rapid preliminary publication forum for research results which may later be published in fuller form elsewhere. The sole responsibility for the content and form of each text rests with its author.

GÖTEBORGS
UNIVERSITET