NetSA Python Documentation

Release 1.5

Carnegie Mellon University

May 24, 2016

CONTENTS

CHAPTER

ONE

NETSA.SCRIPT — THE NETSA SCRIPTING FRAMEWORK

1.1 Overview

The netsa.script module provides a common framework for building SiLK-based analysis scripts. This framework is intended to make scripts re-usable and automatable without much extra work on the part of script authors. The primary concerns of the scripting framework are providing metadata for cataloging available scripts, standardizing handling of command-line arguments (particularly for flow data input), and locating output files.

Here's an example of a simple Python script using the netsa.script framework.

First is a version without extensive comments, for reading clarity. Then the script is repeated with comments explaining each section.

```
#!/usr/bin/env python

# Import the script framework under the name "script".
from netsa import script

# Set up the metadata for the script, including the title, what it
# does, who wrote it, who to ask questions about it, etc.
script.set_title("Sample Framework Script")
script.set_description("""
    An example script to demonstrate the basic features of the
    netsa.script scripting framework.  This script counts the
    number of frobnitzim observed in each hour (up to a maximum
    volume of frobs per hour.)
""")
script.set_version("0.1")
script.set_contact("H. Bovik <hbovik@example.org>")
script.set_authors(["H. Bovik <hbovik@example.org>"])

script.add_int_param("frob-limit",
    "Maximum volume of frobs per hour to observe.",
    default=10)

script.add_float_param("frobnitz-sensitivity",
    "Sensitivity (between 0.0 and 1.0) of frobnitz categorizer.",
    default=0.61, expert=True, minimum=0.0, maximum=1.0)

script.add_flow_params(require_pull=True)
```

```
script.add_output_file_param("output-path",
    "Number of frobnitzim observed in each hour of the flow data.",
    mime_type="text/csv")

# See the text for discussion of the next two functions.

def process_hourly_data(out_file, flow_params, frob_limit, frob_sense):
    ...

def main():
    frob_limit = script.get_param("frob-limit")
    frobnitz_sensitivity = script.get_param("frobnitz-sensitivity")
    out_file = script.get_output_file("output-path")
    for hour_params in script.get_flow_params().by_hour():
        process_hourly_data(out_file, hour_params, frob_limit,
                            frobnitz_sensitivity)

script.execute(main)
```

Let's break things down by section:

```
#!/usr/bin/env python

from netsa import script
```

This is basic Python boilerplate. Any other libraries we use would also be imported at this time.

```
script.set_title("Sample Framework Script")
script.set_description("""
    An example script to demonstrate the basic features of the
    netsa.script scripting framework.  This script counts the
    number of frobnitzim observed in each hour (up to a maximum
    volume of frobs per hour.)
""")
script.set_version("0.1")
script.set_contact("H. Bovik <hbovik@example.org>")
script.set_authors(["H. Bovik <hbovik@example.org>"])
```

Script metadata allows users to more easily find out information about a script, and browse available scripts stored in a central repository. The above calls define all of the metadata that the *netsa.script* framework currently supports. It is possible that a future version will include additional metadata fields.

```
script.add_int_param("frob-limit",
    "Maximum volume of frobs per hour to observe.",
    default=10)

script.add_float_param("frobnitz-sensitivity",
    "Sensitivity (between 0.0 and 1.0) of frobnitz categorizer.",
    default=0.61, expert=True, minimum=0.0, maximum=1.0)
```

Script parameters are defined by calling `netsa.script.add_X_param` (where *X* is a type) for each parameter. Depending on the type of the parameter, there may be additional configuration options (like *minimum* and *maximum* for the float parameter above) available. See the documentation for each function later in this document.

Expert parameters are tuning parameters that are intended for expert use only. An expert parameter is created by setting *expert* to `True` when creating a new parameter. This parameter will then be displayed only if the user asks for `--help-expert`, and the normal help will indicate that expert options are available.

```
script.add_flow_params(require_pull=True)
```

Parameters involving flow data are handled separately, in order to ensure that flows are handled consistently across all of our scripts. The `netsa.script.add_flow_params` function is used to add all of the flow related command-line arguments at once. There is currently only one option. If the *require_pull* option is set, the flow data must come from an `rwfilter` data pull (including switches like `--start-date`, `--end-date`, `--class`, etc.) If *require_pull* is not set, then it is also possible for input files or pipes to be given on the command-line.

```
script.add_output_file_param("output-path",
    "Number of frobnitzim observed in each hour of the flow data.",
    mime_type="text/csv")
```

Every output file (not temporary working file) that the script produces must also be defined using calls to the framework—this ensures that when an automated tool is used to run the script, it can find all of the relevant output files. It's preferable, but not required, for a MIME content-type (like `"text/csv"`) and a short description of the contents of the file be included.

```
def process_hourly_data(out_file, flow_params, frob_limit, frob_sense):
    ...
```

In this example, the `process_hourly_data` function would be expected to use the functions in `netsa.util.shell` to acquire and process flow data for each hour (based on the *flow_params* argument). The details have been elided for simplicity in this example.

```
def main():
    frob_limit = script.get_param("frob-limit")
    frobnitz_sensitivity = script.get_param("frobnitz-sensitivity")
    out_file = script.get_output_file("output-path")
    for hour_params in script.get_flow_params().by_hour():
        process_hourly_data(out_file, hour_params, frob_limit,
                            frobnitz_sensitivity)
```

It is important that no work is done outside the `main` function (which can be given any name you wish). If instead you do work in the body of the file outside of a function, that work will be done whether or not the script has actually been asked to do work. (For example, if the script is given `--help`, it will not normally call your `main` function.) So make sure everything is in here.

```
script.execute(main)
```

The final statement in the script should be a call to `netsa.script.execute`, as shown above. This allows the framework to process any command-line arguments (including producing help output, etc.), then call your `main` function, and finally do clean-up work after the completion of your script.

See the documentation for functions in this module for more details on individual features, including further examples.

1.2 Exceptions

exception `netsa.script.`**`ParamError`**(*param*, *value*, *message*)

This exception represents an error in the arguments provided to a script at the command-line. For example, `ParamError('foo', '2x5', 'not a valid integer')` is the exception generated when the value given for an integer param is not parsable, and will produce the following error output when thrown from a script's `main` function:

```
<script-name>: Invalid foo '2x5': not a valid integer
```

exception netsa.script.**UserError**(*message*)
This exception represents an error reported by the script that should be presented in a standard way. For example, `UserError('your message here')` will produce the following error output when thrown from a script's `main` function:

```
<script-name>: your message here
```

exception netsa.script.**ScriptError**(*message*)
This exception represents an error in script definition or an error in processing script data. This is thrown by some netsa.script calls.

1.3 Metadata Functions

The following functions define "metadata" for the script—they provide information about the name of the script, what the script is for, who to contact with problems, and so on. Automated tools can use this information to allow users to browse a list of available scripts.

netsa.script.**set_title**(*script_title : str*)
Set the title for this script. This should be the human-readable name of the script, and denote its purpose.

netsa.script.**set_description**(*script_description : str*)
Set the description for this script. This should be a longer human-readable description of the script's purpose, including simple details of its behavior and required inputs.

netsa.script.**set_version**(*script_version : str*)
Set the version number of this script. This can take any form, but the standard *major* . *minor* (. *patch*) format is recommended.

netsa.script.**set_package_name**(*script_package_name : str*)
Set the package name for this script. This should be the human-readable name of a collection of scripts.

netsa.script.**set_contact**(*script_contact : str*)
Set the point of contact email for support of this script, which must be a single string. The form should be suitable for treatment as an email address. The recommended form is a string containing:

```
Full Name <full.name@contact.email.org>
```

netsa.script.**set_authors**(*script_authors : str list*)
Set the list of authors for this script, which must be a list of strings. It is recommended that each author be listed in the form described for set_contact.

netsa.script.**add_author**(*script_author : str*)
Add another author to the list of authors for this script, which must be a single string. See set_authors for notes on the content of this string.

1.4 Script Parameters

These calls are used to add parameters to a script. When the script is called from the command-line, these are command-line arguments. When a GUI is used to invoke the script, the params might be presented in a variety of ways. This need to support both command-line and GUI access to script parameters is the reason that they've been standardized here. It's also the reason that you'll find no "add an argument with this arbitrary handler function" here.

If you do absolutely need deeper capabilities than are provided here, you can use one of the basic param types and then do additional checking in the `main` function. Note, however, that a GUI will not aid users in choosing acceptable

values for params defined in this way. Also, make sure to raise `ParamError` with appropriate information when you reject a value, so that the error can be most effectively communicated back to the user.

`netsa.script.`**`add_text_param`**(*name : str*, *help : str*[, *required=False*, *default : str*, *default_help : str*, *expert=False*, *regex : str*])

Add a text parameter to this script. This parameter can later be fetched as a `str` by the script using `netsa.script.get_param`. The *required*, *default*, *default_help*, and *expert* arguments are used by all `add_X_param` calls, but each kind of parameter also has additional features that may be used. See below for a list of these features for text params.

Example: Add a new parameter which is required for the script to run.

```
add_text_param("graph-title",
    "Display this title on the output graph.",
    required=True)
```

It is an error if this parameter is not set, and the script will exit with a usage message when it is run at the command-line.

Example: Add a new parameter with a default value of "" (the empty string):

```
add_text_param("graph-comment",
    "Display this comment on the output graph.",
    default="")
```

If the parameter is not provided, the default value will be used.

Example: Display something different in the help text than the actual default value:

```
add_text_param("graph-date",
    "Display data for the given date.",
    default=date_for_today(), default_help="today")
```

Sometimes a default value should be computed but not displayed as the default to the user when they ask for help at the command-line. In this case, a default value should be provided (which will be displayed to users of a GUI), while a value for *default_help* will be presented in the *–help* output. In addition, GUIs will also display the value of *default_help* in some way next to the entry field for this parameter.

It is perfectly legal to provide a value for *default_help* and not provide a value for *default*. This makes sense when the only way to compute the default value for the field is at actual execution time. (For example, if the end-date defaults to be the same as the provided start-date.)

Example: Add a new "expert" parameter:

```
add_text_param("gnuplot-extra-commands",
    "Give these extra command to gnuplot when writing output.",
    expert=True)
```

Expert parameters are not listed for users unless they explicitly ask for them. (For example, by using `--help-expert` at the command line.)

Other keyword arguments meaningful for text params:

regex Require strings to match this regular expression.

Example: Add a new text parameter that is required to match a specific pattern for phone numbers:

```
add_text_param("phone-number",
    "Send reports to this telephone number.",
    regex=r"[0-9]{3}-[0-9]{3}-[0-9]{4}")
```

netsa.script.**add_int_param**(*name : str*, *help : str*[, *required=False*, *default : int*, *default_help : str*, *expert=False*, *minimum : int*, *maximum : int*])

Add an integer parameter to this script. This parameter can later be fetched as an `int` by the script using `netsa.script.get_param`. The *required*, *default*, *default_help*, and *expert* arguments are described in the help for `netsa.script.add_text_param`.

Other keyword arguments meaningful for integer parameters:

minimum Only values greater than or equal to this value are allowed for this param.

maximum Only values less than or equal to this value are allowed for this param.

Example: Add a new int parameter which is required to be in the range 0 <= x <= 65535.

```
add_int_param("targeted-port",
    "Search for attacks targeting this port number.",
    required=True, minimum=0, maximum=65535)
```

netsa.script.**add_float_param**(*name : str*, *help : str*[, *required=False*, *default : float*, *default_help : str*, *expert=False*, *minimum : float*, *maximum : float*])

Add a floating-point parameter to this script. This parameter can later be fetched as a :class‘float‘ by the script using `netsa.script.get_param`. The *required*, *default*, *default_help* and *expert* arguments are described in the help for `netsa.script.add_text_param`.

Other keyword arguments meaningful for floating-point parameters:

minimum Only values greater than or equal to this value are allowed for this param.

maximum Only values less than or equal to this value are allowed for this param.

netsa.script.**add_date_param**(*name : str*, *help : str*[, *required=False*, *default : datetime*, *default_help : str*, *expert=False*])

Add a date parameter to this script. This parameter can later be fetched by the script as a `datetime.datetime` object using `netsa.script.get_param`. The *required*, *default*, *default_help*, and *expert* arguments are described in the help for `netsa.script.add_text_param`.

netsa.script.**add_label_param**(*name : str*, *help : str*[, *required=False*, *default : str*, *default_help : str*, *expert=False*, *regex : str*])

Add a label parameter to this script. This parameter can later be fetched by the script as a Python `str` using `netsa.script.get_param`. The *required*, *default*, *default_help*, and *expert* arguments are described in the help for `netsa.script.add_text_param`.

Other keyword arguments meaningful for label params:

regex Require strings to match this regular expression, instead of the default `r"[^\S,]+"` (no white space or commas).

Example: Add a new label parameter that is required to match a specific pattern for phone numbers:

```
add_label_param("output-label",
    "Store output to the destination with this label.",
    regex=r"[0-9]{3}-[0-9]{3}-[0-9]{4}")
```

netsa.script.**add_file_param**(*name : str*, *help : str*[, *required=False*, *default : str*, *default_help : str*, *expert=False*, *mime_type : str*])

Add a file parameter to this script. This parameter can later be fetched by the script as a Python `str` filename using `netsa.script.get_param`. The *required*, *default*, *default_help*, and *expert* arguments are described in the help for `netsa.script.add_text_param`.

When the script is run at the command-line, an error will be reported to the user if they specify a file that does not exist, or the path of a directory.

Other keyword arguments meaningful for file params:

mime_type The expected MIME Content-Type of the file, if any.

netsa.script.**add_dir_param**(*name : str*, *help : str*[, *required=False*, *default : str*, *default_help : str*, *expert=False*])

Add a directory parameter to this script. This parameter can later be fetched by the script as a Python `str` filename using `netsa.script.get_param`. The *required*, *default*, *default_help*, and *expert* arguments are described in the help for `netsa.script.add_text_param`.

When the script is run at the command-line, an error will be reported to the user if they specify a directory that does not exist, or the path of a file.

netsa.script.**add_path_param**(*name : str*, *help : str*[, *required=False*, *default : str*, *default_help : str*, *expert=False*])

Add a path parameter to this script. This parameter can later be fetched by the script as a Python `str` using `netsa.script.get_param`. The *required*, *default*, *default_help*, and *expert* arguments are described in the help for `netsa.script.add_text_param`.

netsa.script.**add_path_param**(*name : str*, *help : str*[, *required=False*, *default : str*, *default_help : str*, *expert=False*])

Add a path parameter to this script. This parameter can later be fetched by the script as a Python `str` using `netsa.script.get_param`. The *required*, *default*, *default_help*, and *expert* arguments are described in the help for `netsa.script.add_text_param`.

netsa.script.**add_flag_param**(*name : str*, *help : str*[, *default=False*, *default_help : str*, *expert=False*])

Add a flag parameter to this script. This parameter can later be fetched by the script as a `bool` using `netsa.script.get_param`. The *default*, *default_help*, and *expert* arguments are described in the help for `netsa.script.add_text_param`.

netsa.script.**get_param**(*name : str*) → value

Returns the value of the parameter given by the `str` argument *name*. This parameter will be in the type specified for the param when it was added (for example, date parameters will return a `datetime.datetime` object.) Note that a parameter with no default that is not required may return `None`.

netsa.script.**get_extra_args**() → str list

Returns any extra un-named arguments from the command-line.

1.5 Verbose Output

netsa.script.**get_verbosity**() → int

Returns the current verbosity level (default 0) for the script invocation. The `message` function may be used to automatically emit messages based on the verbosity level set for the script. Verbosity is set from the command-line via the `--verbose` or `-v` flags.

netsa.script.**display_message**(*text*[, *min_verbosity=1*])

Writes the string *text* to `stderr`, as long as the script's verbosity is greater than or equal to *min_verbosity*. Verbosity is set from the command-line via the `--verbose` or `-v` flags. The current verbosity level may be retrieved by using the `get_verbosity` function.

Use this function to write debugging or informational messages from your script for command-line use. For example, writing out which file you are processing, or what stage of processing is in progress.

Do not use it to write out important information such as error messages or actual output. (See `UserError` or `add_output_file_param` and `add_output_dir_param` for error messages and output.)

1.6 Flow Data Parameters

In order to standardize the large number of scripts that work with network flow data using the SiLK tool suite, the following calls can be used to work with flow data input.

netsa.script.**add_flow_annotation**(*script_annotation : str*)
Add a note that will automatically be included in SiLK data pulls generated by this script. This will be included only by rwfilter pulls created by this script using Flow_params.

netsa.script.**add_flow_params**([*require_pull=False*, *without_params : str list*])
Add standard flow parameters to this script. The following params are added by default, but individual params may be disabled by including their names in the *without_params* argument. You might wish to disable the --type param, for example, if your script will run the same pull multiple times, once with --type=in,inweb, then again with --type=out,outweb. (Of course, you might then also want to add in-type and out-type params to the script.)

--class Req Arg. Class of data to process

--type Req Arg. Type(s) of data to process within the specified class. The type names and default type(s) vary by class. Use all to process every type for the specified class. Use *rwfilter –help'* for details on valid class/type pairs.

--flowtypes Req Arg. Comma separated list of class/type pairs to process. May use all for class and/or type. This is alternate way to specify class/type; switch cannot be used with --class and --type

--sensors Req Arg. Comma separated list of sensor names, sensor IDs, and ranges of sensor IDs. Valid sensors vary by class. Use *mapsid* to see a mapping of sensor names to IDs and classes.

--start-date Req Arg. First hour of data to process. Specify date in YYYY/MM/DD[:HH] format: time is in UTC. When no hour is specified, the entire date is processed. Def. Start of today

--end-date Req Arg. Final hour of data to process specified as YYYY/MM/DD[:HH]. When no hour specified, end of day is used unless *start-date* includes an hour. When switch not specified, defaults to value in *start-date*.

If the *require_pull* argument to netsa.script.add_flow_params is not True, input filenames may be specified bare on the command-line, and the following additional options are recognized:

--input-pipe Req Arg. Read SiLK flow records from a pipe: stdin or path to named pipe. No default

--xargs (expert) Req Arg. Read list of input file names from a file or pipe pathname or stdin. No default

The values of these parameters can later be retrieved as a netsa.script.Flow_params object using netsa.script.get_flow_params.

netsa.script.**get_flow_params**() → Flow_params
Returns a Flow_params object encapsulating the rwfilter flow selection parameters the script was invoked with. This object is filled in based on the command-line arguments described in add_flow_params.

class netsa.script.**Flow_params**([*flow_class : str*, *flow_type : str*, *flowtypes : str list*, *sensors : str list*, *start_date : datetime*, *end_date : datetime*, *input_pipe : str*, *xargs : str*, *filenames : str list*])
This object represents the flow selection arguments to an rwfilter data pull. In typical use it is built automatically from command-line arguments by the netsa.script.get_flow_params call. Afterwards, methods such as by_hour are used to modify the scope of the data pull, and then the parameters are included in a call to rwfilter using the functions in netsa.util.shell.

Example: Process SMTP data from the user's requested flow data:

```
netsa.util.shell.run_parallel(
    ["rwfilter %(flow_params)s --protocol=6 --aport=25 --pass=stdout",
     "rwuniq --fields=sip",
     ">>output_file.txt"],
    vars={'flow_params': script.get_flow_params()})
```

Example: Separately process each hour's SMTP data from the user's request flow data:

```
flow_params = script.get_flow_params()
# Iterate over each hour individually
for hourly_params in flow_params.by_hour():
    # Format ISO-style datetime for use in a filename
    sdate = iso_datetime(hourly_params.get_start_date())
    netsa.util.shell.run_parallel(
        ["rwfilter %(flow_params)s --protocol=6 --pass=stdout",
         "rwuniq --fields=dport",
         ">>output_file_%(sdate)s.txt"],
        vars={'flow_params': hourly_params,
              'sdate': sdate})
```

by_day() → Flow_params iter
Given a Flow_params object including a start-date and an end-date, returns an iterator yielding a Flow_params for each individual day in the time span.

If the original Flow_params starts or ends on an hour that is not midnight, the first or last yielded pulls will not be for full days. All of the other pulls will be full days stretching from midnight to midnight.

See also by_hour which iterates over the time span of the Flow_params by hours instead of days.

Raises a ScriptError if the Flow_params has no date information (for example, the script user specified input files rather than a data pull.) This can be prevented by using *require_pull* in your call to script.add_flow_params.

by_hour() → Flow_params iter
Given a Flow_params object including a start-date and an end-date, returns an iterator yielding new Flow_params object identical to this one specialized for each hour in the time period.

Example (strings are schematic of the Flow_params involved):

```
>>> # Note: Flow_params cannot actually take a str argument like this.
>>> some_flows = Flow_params('--type in,inweb --start-date 2009/01/01T00 '
>>>                          '--end-date 2009/01/01T02')
>>> list(some_flows.by_hour())
[netsa.script.Flow_params('--type in,inweb --start-date 2009/01/01T00 '
                          '--end-date 2009/01/01T00'),
 netsa.script.Flow_params('--type in,inweb --start-date 2009/01/01T01 '
                          '--end-date 2009/01/01T01'),
 netsa.script.Flow_params('--type in,inweb --start-date 2009/01/01T02 '
                          '--end-date 2009/01/01T02')]
```

See also by_day which iterates over the time span of the Flow_params by days instead of hours.

Raises a ScriptError if the Flow_params has no date information (for example, the script user specified input files rather than a data pull.) This can be prevented by using *require_pull* in your call to script.add_flow_params.

by_sensor() → Flow_params iter
Given a Flow_params object including a data pull, returns an interator yielding a Flow_params for each individual sensor defined in the system.

get_argument_list() → str list or None
Returns the bundle of flow selection parameters as a list of strings suitable for use as command-line arguments in an `rwfilter` call. This is automatically called by the `netsa.util.shell` routines when a `Flow_params` object is used as part of a command.

get_class() → str or None
Returns the `rwfilter` pull `--class` argument as a `str`.

get_end_date() → datetime or None
Returns the `rwfilter` pull `--end-date` argument as a `datetime.datetime` object.

get_filenames() → str list or None
Returns any files given on the command-line for an `rwfilter` pull as a `str`.

get_flowtypes() → str list or None
Returns the `rwfilter` pull `--flowtypes` argument as a `str`.

get_input_pipe() → str or None
Returns the `rwfilter` pull `--input-pipe` argument as a `str`.

get_sensors() → str list or None
Returns the `rwfilter` pull `--sensors` argument as a `list` of `str`.

get_start_date() → datetime or None
Returns the `rwfilter` pull `--start-date` argument as a `datetime.datetime` object.

get_type() → str or None
Returns the `rwfilter` pull `--type` argument as a `str`.

get_xargs() → str or None
Returns the `rwfilter` pull `--xargs` argument as a `str`.

is_files() → bool
Returns `True` if this `Flow_params` object represents processing of already retrieved files.

is_pull() → bool
Returns `True` if this `Flow_params` object represents a data pull from the repository. (i.e. it contains selection switches.)

using([*flow_class : str*, *flow_type : str*, *flowtypes : str list*, *sensors : str list*, *start_date : datetime*, *end_date : datetime*, *input_pipe : str*, *xargs : str*, *filenames : str list*]) → Flow_params
Returns a new `Flow_params` object in which the arguments in this call have replaced the parameters in *self*, but all other parameters are the same.

Raises a `ScriptError` if the new parameters are inconsistent or incorrectly typed.

1.7 Producing Output

Every output file that a script produces needs to be registered with the system, so that automated tools can be sure to collect everything. Some scripts produce one or more set outputs. For example "the report", or "the HTML version of the report". Others produce a number of outputs based on the content of the data they process. For example "one image for each host we identify as suspicious."

netsa.script.**add_output_file_param**(*name : str*, *help: str*[, *required=True*, *expert=False*, *description : str*, *mime_type='application/octet-stream'*])
Add an output file parameter to this script. This parameter can later be fetched by the script as a Python `str` filename or a Python `file` object using `netsa.script.get_output_file_name` or `netsa.script.get_output_file`. Note that if you ask for the file name, you may wish to handle the filenames `stdout`, `stderr`, and `-` specially to be consistent with other tools. (See the documentation of `netsa.script.get_output_file_name` for details.) Output file parameters are required by default. If

an output file parameter is not required, the implication is that if the user does not specify this argument, then this output is not produced.

You should probably not use default values for output file parameters other than `"stdout"` and `"stderr"`.

In keeping with the behavior of the SiLK tools, it is an error for the user to specify an output file that already exists. If the environment variable `SILK_CLOBBER` is set, this restriction is relaxed and existing output files may be overwritten.

The *mime_type* argument is advisory., but it should be set to an appropriate MIME content type for the output file. The framework will not report erroneous types, nor will it automatically convert from one type to another. Examples:

`text/plain` Human readable text file.

`text/csv` Comma-separated-value file.

`application/x-silk-flows` SiLK flow data

`application/x-silk-ipset` SiLK ipset data

`application/x-silk-bag` SiLK bag data

`application/x-silk-pmap` SiLK prefix map data

`image/png` etc. Various standard formats, many of which are listed on IANA's website.

It is by no means necessary to provide a useful MIME type, but it is helpful to automated systems that wish to interpret or display the output of your script.

The *description* argument may also be provided, with a long-form text description of the contents of this output file. Note that *description* describes the contents of the file, while *help* describes the meaning of the command-line argument.

`netsa.script.`**`get_output_file_name`**(*name : str*) → str

Returns the filename for the output parameter *name*. Note that many SiLK tools treat the names `stdout`, `stderr`, and `-` as meaning something special. `stdout` and `-` imply the output should be written to standard out, and `stderr` implies the output should be written to standard error. It is not required that you handle these special names, but it helps with interoperability. Note that you may need to take care when passing these filenames to SiLK command-line tools for output or input locations, for the same reason.

If you use `netsa.script.get_output_file`, it will automatically handle these special filenames.

If this output file is optional, and the user has not specified a location for it, this function will return `None`.

`netsa.script.`**`get_output_file`**(*name : str*) → file

Returns an open `file` object for the output parameter *name*. The special names `stdout`, `-` are both translated to standard output, and `stderr` is translated to standard error.

If you need the output file name, use `netsa.script.get_output_file_name` instead.

If *append* is `True`, then the file is opened for append. Otherwise it is opened for write.

`netsa.script.`**`add_output_dir_param`**(*name : str*, *help : str*[, *required=True*, *expert=False*, *description : str*, *mime_type : str*])

Add an output directory parameter to this script. This parameter can later be used to construct a `str` filename or a Python `file` object using `netsa.script.get_output_dir_file_name` or `netsa.script.get_output_dir_file`. Unlike most parameters, output directory parameters never have default values, and are required by default. If an output directory parameter is not required, the implication is that if the user does not specify this argument, then this output is not produced.

See `add_output_file_param` for the meanings of the *description* and *mime_type* arguments. In this context, these arguments provide default values for files created in this output directory. Each individual file can be given its own *mime_type* and *description* when using the

netsa.script.get_output_dir_file_name and netsa.script.get_output_dir_file functions.

netsa.script.**get_output_dir_file_name**(*dir_name : str*, *file_name : str*[, *description : str*, *mime_type : str*]) → str

Returns the path for the file named *file_name* in the output directory specified by the parameter *dir_name*. Also lets the netsa.script system know that this output file is about to be used. If provided, the *description* and *mime_type* arguments have meanings as described in add_output_file_param. If these arguments are not provided, the defaults from the call where *dir_name* was defined in add_output_dir_param are used.

If the output directory parameter is optional, and the user has not specified a location for it, this function will return None.

netsa.script.**get_output_dir_file**(*dir_name : str*, *file_name : str*[, *description : str*, *mime_type : str*]) → file

Returns the an open file object for the file named *file_name* in the output directory specified by the parameter *dir_name*. Also lets the netsa.script system know that this output file is about to be used. If provided, the *description* and *mime_type* arguments have meanings as described in add_output_file_param. If these arguments are not provided, the defaults from the call where *dir_name* was defined in add_output_dir_param are used.

If the output dir param is optional, and the user has not specified a location for it, this function will return None.

If *append* is True, the file is opened for append. Otherwise, the file is opened for write.

1.8 Script Execution

netsa.script.**execute**(*func : callable*)

Executes the main function of a script. This should be called as the last line of any script, with the script's main function (whatever it might be named) as its only argument.

It is important that all work in the script is done within this function. The script may be loaded in such a way that it is not executed, but only queried for metadata information. If the script does work outside of the main function, this will cause metadata queries to be very inefficient.

CHAPTER

TWO

NETSA.SCRIPT.GOLEM — GOLEM SCRIPT AUTOMATION

2.1 Overview

The *Golem Script Automation* framework is a specialized extension of the *NetSA Scripting Framework* for constructing automated analysis scripts. Such scripts might be launched periodically from a cron job in order to produce regular sets of results in a data repository. In addition to the golem-specific extensions, `netsa.script.golem` offers the same functionality as `netsa.script`.

At its heart, Golem is a template engine combined with some synchronization logic across analytic time bins. The templates define the output paths for result data and the command line sequences and pipelines used to generate these results. Template variables are known as *tags*. Golem provides certain tags by default and others can be added or modified via the golem *configuration functions*.

Golem Automation is designed to allow developers to build scripts that easily consume the output results of other golem scripts without the need for detailed knowledge of the implementation details of the external script (e.g. how often it runs or what pathnames it uses for populating its data repository). When provided the path to the external script in its configuration, a golem script will interrogate the external script for information regarding its output results, automatically synchronize across processing windows, and make the result paths available for use in command templates as input paths.

In addition to analysis automation, Golem scripts also offer a query mode so that results in the data repository can be easily examined or pulled to local directories.

Golem offers a number of shortcuts and convenience functions specific to the SiLK Flow Analysis suite, but is not limited to using SiLK for analysis.

See the *examples* at the end of this chapter to learn how to write a golem script. See the description of *template and tag usage* for details on tags provided for use within templates. See the *API reference* for more thorough documentation of the features and interface. See the *CLI reference* for the standard command line parameters that golem enables.

2.2 Command Line Usage

Golem-enabled scripts offer standard command line parameters grouped into three categories: *basic*, *repository-related*, and *query-related*. Parameters must be enabled by the script author. Parameters can be enabled individually or by category. The `add_golem_params` function will enable all parameters.

Golem scripts also have the standard `netsa.script` command line options `--help`, `--help-expert`, and `--verbose`.

2.2.1 Basic Parameters

The following golem command line parameters control the filtering of processing windows, as well as some input and output behavior. These options affect both repository and query operations.

-last-date <date>
: Date specifying the last time bin of interest. The provided date will be rounded down to the first date of the processing interval in which it resides. Default: most recent

-first-date <date>
: Date specifying the first time bin of interest. The provided date will be rounded down to the first date of the processing interval in which it resides. Default: value of *--last-date*

-intervals <count>
: Process or query the last *count* intervals (as defined by `set_interval` within the script header) of data from the current date. This will override any values provided by *--first-date* or *--last-date*.

-skip-incomplete
: Skip processing intervals that have incomplete input requirements (i.e. ignore the date if any source dependencies have incomplete results).

-overwrite
: Overwrite output results if they already exist.

-<loop-select>
: Each template loop that has been defined in the golem script (via the `add_loop` function) has an associated parameter that allows a comma-separated selection of a subset of that loop's values. For example, if a sensor loop was defined under the tag `sensor`, the parameter would be `--sensor` by default. If grouping was enabled it would be `--sensor-group` instead.

2.2.2 Repository Parameters

Repository parameters control the core data processing and generation of data in the result repository, typically run via a cron job.

-data-process
: This is the main parameter that makes the script do its work. It generates and stores incomplete or missing analysis results in the data repository, skipping those that are complete (unless *--overwrite* is given).

-data-status
: Show the status of repository processing bins, for the provided options. No processing is performed.

-data-queue
: List the dates of all pending repository results for the provided options. No processing is performed.

-data-complete
: List the dates of all completed repository results, for the provided options. No processing is performed.

-data-inputs
: Show the status of input dependencies from other golem scripts or defined templates, for the provided options. Use `-v` or `-vv` for less abbreviated paths. No processing is performed.

-data-outputs
: Show the status of repository output results for the provided options. Use `-v` or `-vv` for less abbreviated paths. No processing is performed.

2.2.3 Query Parameters

Query parameters control how local copies of results are stored, both when copying from the repository or when performing a fresh analysis.

-output-path <path>
: Generate a single query result in the specified output file for the given parameters. Also accepts '-' and 'stdout'.

-output-select <arg1[,arg2[,...]]>
: For golem scripts that have more than one output defined, limit output results to the comma-separated names provided.

-output-dir <path>
: Copy query result files in the specified output directory for the given parameters. The files will follow the same naming scheme as specified for the repository. These names can be previewed via the *--show-outputs* option.

-show-inputs
: Show the status of input dependencies from other golem scripts or defined templates for the provided options. Use -v or -vv for less abbreviated paths. No processing is performed.

-show-outputs
: Show relative output paths that would be generated within *--output-dir*, given the options provided. Use -v or -vv for less abbreviated paths. No processing is performed.

2.3 Metadata Functions

Golem scripts are an extension of netsa.script. As such, golem scripts offer the same functions as netsa.script. Script authors are encouraged to use the following *metadata functions*:

netsa.script.golem.**set_title**(*script_title : str*)
: Set the title for this script. This should be the human-readable name of the script, and denote its purpose.

netsa.script.golem.**set_description**(*script_description : str*)
: Set the description for this script. This should be a longer human-readable description of the script's purpose, including simple details of its behavior and required inputs.

netsa.script.golem.**set_version**(*script_version : str*)
: Set the version number of this script. This can take any form, but the standard *major* . *minor* (. *patch*) format is recommended.

netsa.script.golem.**set_package_name**(*script_package_name : str*)
: Set the package name for this script. This should be the human-readable name of a collection of scripts.

netsa.script.golem.**set_contact**(*script_contact : str*)
: Set the point of contact email for support of this script, which must be a single string. The form should be suitable for treatment as an email address. The recommended form is a string containing:

```
Full Name <full.name@contact.email.org>
```

netsa.script.golem.**set_authors**(*script_authors : str list*)
: Set the list of authors for this script, which must be a list of strings. It is recommended that each author be listed in the form described for set_contact.

netsa.script.golem.**add_author**(*script_author : str*)
: Add another author to the list of authors for this script, which must be a single string. See set_authors for notes on the content of this string.

Please see netsa.script for additional functions, such as those used to *add custom command line parameters*.

2.4 Configuration Functions

Golem scripts are configured by calling functions from within a particular imported module. You can either import `netsa.script.golem` directly:

```
from netsa.script import golem
```

Or import as `netsa.script.golem.script`:

```
from netsa.script.golem import script
```

In either case, the imported module will provide identical functionality. All functions available within `netsa.script` are also available from within `golem`, with the addition of the golem-specific functions and classes. Please consult the `netsa.script` documentation for details on functions offered by that module.

The following functions configure the behavior of golem scripts.

`netsa.script.golem.`**`set_default_home`**(*path : str*[, *path : str*, ...])
Sets the default base path for this golem script in cases where the `GOLEM_HOME` environment variable is not set. Multiple arguments will be joined together as with `os.path.join`. If the provided path is relative, it is assumed to be relative to the directory in which script resides.

The actual home path will be decided by the first available source in the following order:

1. The `GOLEM_HOME` environment variable
2. The default home if set by this function
3. The directory in which the script resides

Subsequent path settings (e.g. `set_repository`) will be relative to the script home if they are not absolute paths.

`netsa.script.golem.`**`set_repository`**(*path : str*[, *path : str*, ...])
Sets the default path for the output results data repository. Multiple arguments will be joined together as with `os.path.join`. Output results will be stored in this directory or in a subdirectory beneath, depending on how each output template is specified. Relative paths are considered to be relative to the `home` path.

`netsa.script.golem.`**`set_suite_name`**(*name : str*)
Set the short suite name for this script, if it belongs to a suite of related scripts. This should be a simple label suitable for use as a component in paths or filenames. Defaults to `None`.

`netsa.script.golem.`**`set_name`**(*name : str*)
Set the short name for this script. The name should be a simple label suitable for use as a component in paths or filenames. This defaults to the basename of the script file itself, minus the '.py' extension, if present.

`netsa.script.golem.`**`set_interval`**([*days : int*, *minutes : int*, *hours : int*, *weeks : int*])
Set how often this golem script is expected to generate results.

The interval roughly corresponds to how often the script should be run (such as from a cron job). Golem scripts will only process data for incomplete intervals over a provided date range, unless told otherwise via the *`--overwrite`* option.

`netsa.script.golem.`**`set_span`**([*days : int*, *minutes : int*, *hours : int*, *weeks : int*])
Set the span over which this golem script will expect input data for each processing interval. Defaults to the processing interval.

The span will manifest as how much data is being pulled from a SiLK repository or possibly how many outputs from another golem script are being consumed. For example, a script having a 4 week span might run once a week, pulling 4 weeks worth of data each time it runs.

See *Intervals and Spans Explained* for more details.

netsa.script.golem.**set_skew**([*days : int*, *minutes : int*, *hours : int*, *weeks : int*])
Shift the epoch from which all time bins and spans are anchored. For example, given a golem script with an interval of one week, skew can control which day of the week processing occurs.

netsa.script.golem.**set_lag**([*days : int*, *minutes : int*, *hours : int*, *weeks : int*])
Set the lag for this golem script relative to the current date and time. Defaults to 0.

As an example, 3 hours is a typical value for data to finish accumulating in a given hour within a SiLK repository. Setting lag to 3 hours effectively shifts the script's concept of the current time that far into the past.

netsa.script.golem.**set_realtime**(*enable=True*)
Set whether or not this golem script will report output results in real time or not. Defaults to `False`.

Normally a golem script will wait until a processing interval has completely passed before performing any processing or reporting any output results. With `set_realtime` enabled, the golem script will consider the current processing bin to be the most recent, even if it extends to a future date.

Enabling realtime has a side effect of setting *lag* to zero.

netsa.script.golem.**set_tty_safe**(*enable=True*)
Set whether or not query results are safe to send to the terminal. Defaults to `False`.

netsa.script.golem.**set_passive_mode**(*enable=False*)
Controls whether or not an output option is required before running the main loop of the script. Defaults to `False`. Normally at least one repository-related or query-related option must be present or the script aborts. This is useful for scripts that require query behavior by default or are maintaining the repository in a custom fashion. If enabled, script authors should explicitely check whether or not repository updates were requested via the command line prior to updating the repository.

netsa.script.golem.**add_tag**(*name : str*, *value : str or func*)
Set a command template tag with key *name*. The provided value can be callable, in which case it is resolved once the `main` function is invoked. Tags can reference other tags. See *template and tag usage* for more information.

netsa.script.golem.**add_loop**(*name : str*, *value : str list or func*[, *group_by : str or func*, *group_name : str*, *sep=', '*])
Add a template tag under key *name* whose values cycle through those provided by *values*, either as an iterable or callable. In the latter case, the values are not resolved until the `main` function is invoked.

Optional keyword arguments:

group_by Specifies how to group entries from *values* into a single loop entry. If the provided value is callable, the function should accept a single entry from *values* and return the group label for that entry or the original string, e.g. 'LAB2' might return 'LAB'. If the value is a dictionary, it will resolve to the mapped value if present. If it is an iterable of prefix matches, they will be converted into a regular expression to be applied to the beginning of each entry. In all cases, if there is no match or result, the original entry becomes its own group label.

group_name This is the name of the template tag under which the group label appears, defaulting to the value of *name* appended with '_group'. In the above example, if *name* is 'sensor', then `%(sensor)s` might resolve to 'LAB1,LAB2,LAB3' whereas `%(sensor_group)s` would merely resolve to 'LAB'.

sep Depending on how these loops are visited, the template value under *name* might contain multiple values from *values*. Under these circumstances, the resulting string is joined by the value of 'sep' (default: ',')

Note that adding a loop will automatically add an additional query command line parameter named after the *name* for limiting which values to process within the loop. If grouping was requested, an additional parameter named after the *group_name* is also added. If these parameters are not desired or need to be modified, use the `modify_golem_param` function.

See *template usage* for more information on templates and loop values.

netsa.script.golem.**add_sensor_loop**(*[name='sensor' : str, sensors : str list or func, group_by : str or func, group_name : str, auto_group=False]*)

This is a convenience function for adding a template loop tag under key *name* whose values are based on sensors defined in a SiLK repository. Special note is taken that these loop values represent SiLK sensors, so any netsa.script.Flow_params tags that are defined (see add_flow_tag) will have their *sensors* parameter automatically bound (if not explicitly bound to something else) to the last sensor loop defined by this function.

Optional keyword arguments:

name The tag name within the template. Defaults to 'sensor', accessible from within templates as the tag %(sensor)s

sensors The source of sensor names, specified either as an iterable or callable. Defaults to the get_sensors function, which will interrogate the local SiLK repository once the main function is invoked.

group_by Same as with add_loop

group_name Same as with add_loop

auto_group Causes *group_by* to be set to the get_sensor_group function, a convenience function that strips numbers, possibly preceded by an underscore, from the end of sensor names. This can provide serviceable sensor grouping for descriptive sensor names (e.g. 'LAB0', 'LAB1', 'LAB2') but will not be of much use if they are generically named (e.g. 'S0', 'S1', etc).

netsa.script.golem.**add_flow_tag**(*name : str[, flow_class : str, flow_type : str, flowtypes : str, sensors : str, start_date : str, end_date : str, input_pipe : str, xargs : str, filenames : str]*)

Add a netsa.script.Flow_params object as a template tag under key *name*. The rest of the keyword arguments correspond to the same parameters accepted by the netsa.script.Flow_params constructor and serve to map these fields to *either* specific template tags in the golem script or to the literal string if it is not present in the template tags.. If not otherwise specified, the *start_date* and *end_date* attributes are bound to the values of %(golem_start_date)s and %(golem_end_date)s, respectively, for each loop iteration. Additionally, if any sensor-specific loops were specified via add_sensor_loop, the *sensors* parameter defaults to the tag associated with the last defined sensor loop (typically %(sensor)s). The resulting netsa.script.Flow_params object is associated with a tag entry specified by *name*. The values of tags associated with netsa.script.Flow_params attributes in this way are still accessible under their original tag names.

The following optional keyword arguments are available to map template tag values to their corresponding attributes in the flow params object: *flow_class*, *flow_type*, *flowtypes*, *sensors*, *start_date*, *end_date*, *input_pipe*, *xargs*, and *filenames*.

See *template and tag usage* for more information on templates.

netsa.script.golem.**add_output_template**(*name : str, template : str[, scope : int, mime_type : str, description : str]*)

Define an output template tag key *name* with the provided *template*. The provided template can use any of the tags available for each processing iteration. Wildcards are acceptable in the template specification. Absolute paths are not allowed since outputs must reside in the data repository.

The following optional keyword arguments are available. Pass any of them a value of None to disable entirely:

scope Defines how many intervals of this output are required to represent a complete analysis result in cases where the output from a single interval represents a partial result. For example, a golem script might have an interval of 1 day whereas a "complete" set of results is 7 days worth relative to any particular day.

mime_type The expected MIME Content-Type of the output file, if any.

description A long-form text description, if any, of the contents of this output file.

See *template and tag usage* for more information on templates.

Templates are not required to reference all distinguishing tags and can therefore be 'lossy' across loop iterations if such a thing is desired.

`netsa.script.golem.`**`add_input_template`**(*name : str*, *template : str*[, *required=True*, *mime_type : str*, *descriptions : str*])

Define an input template tag under key *name*. This is useful for defining inputs not produced by other golem scripts. Wildcards are allowed in the template specification.

The provided template can use the any of the set of tags available on each processing iteration. The resolved string is available to command templates under key *name*.

Optional keyword arguments:

required When `False` will ignore missing inputs once the template is resolved. Defaults to `True`.

count Specifies how many intervals (as defined by this script) over which to resolve the input template. For example, if this script has an *interval* of one week, a *count* of 4 will resolve to the last 4 days of the week in question.

offset Specify how far backwards (in intervals) to anchor this input. If a *count* has been specified, the offset shifts the entire count of intervals.

cover If `True`, a *count* will be calculated such that the template will be resolved over all *intervals* covered by the *span* of this script. For example, a script with an interval of one day and span of seven, the input template will be resolved for all seven days in the span.

source_name Specify what produced this input For informational purposes when listing script inputs.

mime_type The expected MIME Content-Type of the input.

description A long-form text description of the expected contents of this input.

See *template and tag usage* for more information on templates.

`netsa.script.golem.`**`add_golem_input`**(*golem_script : str, name : str [, output_name : str, count : int, cover=False, offset : int, span : timedelta, join_on : str or str list, join : dict, required=True*)

Specify a tag, under key *name*, that represents the path (or paths) to an output of an external *golem script*. Only a single output can be assosciated at at time – if the external script has multiple outputs defined, then additional calls to this function are necessary for each output of interest.

For each output template, efforts are made to synchronize across time intervals and loop tags as appropriate. By default, this is the output of the most recent interval of the other golem script that corresponds to the interval currently under consideration within the local script. Covering multiple intervals of the input is controlled by *cover*, *count*, or *span*. If, for example, you're pulling inputs with an hourly interval into a script with a daily interval, the *cover* parameter should probably be used, otherwise only a single hour would be pulled in.

Matching loop tags are automatically joined unless the *join* parameter is provided, in which case the only joins that happen are the ones provided.

Optional keyword arguments:

output_name The tag name used for this output in the external script if it differs from the value of *name* used locally for this input. By default the names are assumed to be identical.

count Specifies how many intervals (as defined by the external script) of output data are to be used as input. By default, the most recent interval of the other golem script that corresponds to the local interval currently under consideration is provided.

For example, if the other golem script has an *interval* of one week, a *count* of 4 will provide the last 4 weeks of output from that script.

offset Specify how far back (in units of the other script's interval) to reference for this input. Defaults to the most recent corresponding interval. If a *count* has been specified, the offset shifts the entire count of intervals.

For example, if the other script's interval is one week, an offset of -1 will reference the output from the week prior to the most recent week. Negative and positive offsets are equivalent for these purposes, they always reach backwards through time.

cover If `True`, a *count* is calculated that will fully cover the local interval under consideration. This option cannot be used with the *count* or *offset* options. Defaults to `False`.

For example, if the local interval is one week, and the other interval is one day, then this is equivalent to specifying a *count* of 7 (days).

span A `datetime` object that represents a span of time that covers the intervals of interest. Based on the other script's interval, a *count* is calculated that will cover the provided span. Cannot be used simultaneously with *count*, *offset*, or *cover*.

join A dictionary or iterable of tuples that provides an equivalence mapping between template loops defined in the other golem script and locally defined loops. If no join mapping is provided, an attempt is made to join on loops sharing the same name. If the join parameter is provided, no auto-join is performed.

For example, if the other script defines a template loop on the `%(my_sensor)s` tag, and the local script defines a loop on `%(sensor)s`, a mapping from 'my_sensor' to 'sensor' will ensure that for each iteration over the values of `%(sensor)s` the input tag value is also sensor-specific based on its `%(my_sensor)s` loop. Without this association, the input tag would resolve to all outputs across all sensors, regardless of which sensor or sensor group was currently under local consideration. Iterations with no valid mapping are ignored, as opposed to when a valid association exists but the other output is missing.

required If `False` or `0` and the expected output from the other golem script is missing, continue processing rather than raising an exception. If specified as a positive integer, it means *at least* that many of the other inputs should exist, otherwise an exception is thrown. By default, at least one input is required. It is up to the developer to handle missing inputs (i.e. empty template tags) appropriately in these cases.

`netsa.script.golem.`**`add_input_group`**(*name : str*, *group : list*)

Group the given inputs under the provided tag as a single `GolemArgs` object. The result can be used the same as the regular inputs.

`netsa.script.golem.`**`add_output_group`**(*name : str*, *group : list*)

Group the given outputs under the provided tag as a single `GolemArgs` object. The result can be used the same as the regular outputs, including as inputs to other golem scripts.

`netsa.script.golem.`**`add_query_handler`**(*name : str*, *query_handler : func*)

Define a query handler under tag key *name* to be processed by the callable *query_handler*. The output will only be generated dynamically when specifically requested via *query-related parameters*.

The provided callable will be passed the name of this query and a 'tags' dictionary for use with templates. And additonal tag `%(golem_query_tgt)s` will be provided in the standard dictionary that contains the output path for this query. The function is responsible for creating this output.

`netsa.script.golem.`**`add_self_input`**(*name : str*, *output_name : str*[, *count : int*, *offset : int*, *span : timedelta*])

Specify an input template tag, under key *name*, associated with this script's own output from prior intervals. Since output tag names will necessarily collide, *name* is the new tag name for the input and *output_name* is the name of the output template.

Optional keyword arguments:

count Same as with `add_golem_input`

offset Same as with `add_golem_input`, except defaults to -1. If self-referencing for purposes of delta-encoding, 0 should probably be specified.

span Same as with `add_golem_input`

Assuming a local loop over the template tag `%(sensor)s`, the following example:

```
>>> script.add_self_input('result', 'prior_result')
```

...is equivalent to this:

```
>>> script.add_golem_input(script.get_script_path(), 'prior_result',
>>>     output_name='result',
>>>     offset=-1,
>>>     join_on=['sensor'],
>>>     required=False)
```

Note: If this output happens to have a *scope* the developer should ensure that this self-reference means what is intended for 'most recent' output.

For example, if this golem script has an *interval* of 1 day, but the output has a *scope* of 28 (days), the example above would capture the prior 28 days of output due to the offset of -1.

If the script is delta-encoding its current result with its own prior results, however, probably what is desired would be the prior 27 days, in which case offset should be specified as 0. The 28th day in this scope scenario is the very result currently being generated, which does not exist yet, and therefore will not appear in the template as a 'prior' result. After processing is complete, however, it *will* appear in the collected outputs if a different golem script references this interval as input.

`netsa.script.golem.`**`add_golem_source`**(*path : str*)
Adds a directory within which to search for other golem scripts. Directories are searched in the following order: 1) paths within the colon-separated list of directories in the `GOLEM_SOURCES` environment variable; 2) any paths added by this function (multiple invocations allowed); 3) this script's own directory as reported by the `get_script_dir` function, and finally 4) this script's home directory as reported by the `get_home` function, if different than the script directory. These directories are only searched if the external script is specified as a relative path.

`netsa.script.golem.`**`get_script_path`**() → str
Returns the normalized absolute path to this script.

`netsa.script.golem.`**`get_script_dir`**() → str
Returns the normalized absolute path to the directory in which this script resides.

`netsa.script.golem.`**`get_home`**() → str
Returns the current value of this script's home path if it has been set. Otherwise, defaults to the contents of the `GOLEM_HOME` environment variable (if set), or finally, the directory in which this script resides.

`netsa.script.golem.`**`get_repository`**() → str
Returns the current path for this script's data output repository, or `None` if not set.

2.5 Parameter Functions

The following functions are used to enable and modify the standard command-line parameters available for golem scripts. No command line parameters are enabled by default, so at least one of these should be invoked in a typical

golem script:

netsa.script.golem.**add_golem_params**([*without_params : str list*])
Enables all golem command line parameters. Equivalent to individually invoking each of add_golem_basic_params, add_golem_repository_params, and add_golem_query_params. Optionally accepts a list of parameters to exclude.

netsa.script.golem.**add_golem_basic_params**([*without_params : str list*])
Enables *basic* golem command line parameters. Optionally accepts a list of parameters to exclude.

netsa.script.golem.**add_golem_query_params**([*without_params : str list*])
Enables *query-related* golem command line parameters. Optionally accepts a list of parameters to exclude.

netsa.script.golem.**add_golem_repository_params**([*without_params : str list*])
Adds *repository-related* golem command line parameters. Optionally accepts a list of parameters to exclude.

netsa.script.golem.**add_golem_param**(*name : str*[, *alias : str*])
Enables a particular golem command line parameter. Accepts the same optional keyword parameters as the modify_golem_param function with the exception of enable, which is implied. An example is the alias parameter, which which can be provided to change the default parameter string (for example, aliasing --last-date to --date in cases where date ranges are not desired).

netsa.script.golem.**modify_golem_param**(*name : str*[, *enabled : bool*, *alias : str*, *help : str*, ...])
Modifies the settings for the given golem script parameter. Accepts new values for the golem-specific keywords *enabled* and *alias*, along with the usual netsa.script parameter keywords (e.g. *help*, *required*, *default*, *default_help*, *description*, *mime_type*)

2.6 Processing and Status Functions

The following functions are intended for use within the main function during processing or examination of status.

netsa.script.golem.**execute**(*func*)
Executes the main function of a golem script. This should be called as the last line of any golem script, with the script's main function (whatever it might be named) as its only argument.

> **Warning:** It is important that most, if not all, actual work the script does is done within this function. Golem scripts (as with all *NetSA Scripting Framework* scripts) may be loaded in such a way that they are not executed, but merely queried for metadata instead. If the golem script performs significant work outside of the main function, metadata queries will no longer be efficient. Golem scripts must use this execute function rather than netsa.script.execute.

netsa.script.golem.**process**([*golem_view : GolemView*])
Returns a GolemProcess wrapper around the given golem view, which defaults to the main script view. The result behaves much like a GolemTags object. Iterating over it returns a dictionary of resolved template tags while performing system level interactions (such as checking for input existence and creating output directories and/or files) in preparation for whatever processing the developer specifies in the processing loop. Views that have already completed processing are ignored.

Also takes an optional 'exception_handler' keyword argument which must be a function that accepts an exception and a golem view object (advanced use only).

netsa.script.golem.**loop**([*golem_view : GolemView*])
Returns a GolemTags view of the given golem view, which defaults to the main script view. No system level processing happens while iterating over or interacting with this view.

netsa.script.golem.**inputs**([*golem_view : GolemView*])
Returns a GolemInputs view of the given golem view, which defaults to the main script view. No system level processing happens while iterating over or interacting with this view.

netsa.script.golem.**outputs**([*golem_view : GolemView*])
Returns a GolemOutputs view of the given golem view, which defaults to the main script view.. No system level processing happens while iterating over or interacting with this view.

netsa.script.golem.**is_complete**([*golem_view : GolemView*])
Examines the outputs of the optionally provided GolemView object, which defaults to the main script view, and examines the status of the outputs for each processing interval. If all appear to be complete, returns True, otherwise False.

netsa.script.golem.**script_view**()
Returns the currently defined global GolemView object.

netsa.script.golem.**current_view**([*golem_view : GolemView*])
Returns a version of the given GolemView object, which defaults to the main script view, for the most recent interval available.

2.7 Utility Functions

The following functions provide some SiLK-specific tools and other potentially useful features for script authors.

netsa.script.golem.**get_sensors**() → str list
Retrieves a list of sensors as defined in the local SiLK repository configuration.

netsa.script.golem.**get_sensor_group**(*sensor : str*) → str
Convenience function, such as can be passed as a value for the *group_by* parameter in add_sensor_loop, for extracting a sensor 'group' out of a sensor name. Groups are determined by extracting prefixes made of 'word' characters excluding '_'. For example, two sensors called 'LAB0' and 'LAB1' would be grouped under 'LAB'.

netsa.script.golem.**get_sensors_by_group**([*grouper : func*, *sensors : str list*])
Convenience function that uses the callable *grouper* to construct a tuple of named pairs of the form (group_name, members) suitable for use in constructing a dictionary. Defaults to the get_sensor_group function.

netsa.script.golem.**get_args**() → GolemArgs
Returns a GolemArgs object containing any non-parameter arguments that were provided to the script on the command line.

2.8 Additional Functions

Please see the following sections in the netsa.script documentation for details regarding other functions available within netsa.script.golem:

- *Adding Script Parameters*
- *Verbose Output*
- *Flow Data Parameters*
- *Adding Additional Output*

2.9 Usage of Tags, Loops, and Templates

Golem is a templating engine. Based on a script's configuration, the processing loop will iterate over time bins (processing intervals) and the values of any other loops defined for tags in the script. For each iteration, a dictionary of template tags is produced for use with such utilities as `netsa.util.shell`. A typical golem script looks something like this:

```
#!/usr/bin/env python

from netsa.script import golem
from netsa.util import shell

# golem configuration and command line parameter
# configuration up here

def main():

    # All 'real' work should happen in this function.
    # Before invoking the processing loop, perhaps do some
    # configuration and prep work

    for tags in golem.process():

        # Set up per-iteration prep work, such as perhaps
        # some temp files

        # maybe modify contents of tags
        tags['temp_file'] = ...

        # set up command templates

        cmd1 = ...
        cmd2 = ...
        ...

        # run the commands
        shell.run_parallel(cmd1, cmd2, vars=tags)

# pass main function to module for invocation and handling
golem.execute(main)
```

The following template tags are automatically available to each iteration over a golem processing loop:

Tag Name	Contents
`golem_name`	golem script name
`golem_suite`	suite name, if any
`golem_span`	timedelta obj for span
`golem_interval`	timedelta obj for interval
`golem_span_iso`	iso string repr of `golem_span`
`golem_interval_iso`	iso string repr of `golem_interval`
`golem_repository`	data directory for this script
`golem_bin_date`	start datetime for this interval
`golem_bin_year`	interval datetime component 'year'
`golem_bin_month`	interval datetime component 'month'

Continued on next page

Table 2.1 – continued from previous page

Tag Name	Contents
golem_bin_day	interval datetime component 'day'
golem_bin_hour	interval datetime component 'hour'
golem_bin_second	interval datetime component 'second'
golem_bin_microsecond	interval datetime component 'microsecond'
golem_bin_iso	iso string repr for golem_bin_date
golem_bin_basic	iso basic string repr for golem_bin_date
golem_bin_silk	silk string repr for golem_bin_date
golem_start_date	start datetime for this span
golem_start_year	start datetime component 'year'
golem_start_month	start datetime component 'month'
golem_start_day	start datetime component 'day'
golem_start_hour	start datetime component 'hour'
golem_start_second	start datetime component 'second'
golem_start_microsecond	start datetime component 'microsecond'
golem_start_iso	iso string repr for golem_start_date
golem_start_basic	iso basic string repr for golem_start_date
golem_start_silk	silk string repr for golem_start_date
golem_end_date	end datetime for this span
golem_end_year	end datetime component 'year'
golem_end_month	end datetime component 'month'
golem_end_day	end datetime component 'day'
golem_end_hour	end datetime component 'hour'
golem_end_second	end datetime component 'second'
golem_end_microsecond	end datetime component 'microsecond'
golem_end_iso	iso string repr for golem_end_date
golem_end_basic	iso basic string repr for golem_end_date
golem_end_silk	silk string repr for golem_end_date
golem_next_bin_date	next bin datetime for this span
golem_next_bin_year	next bin datetime component 'year'
golem_next_bin_month	next bin datetime component 'month'
golem_next_bin_day	next bin datetime component 'day'
golem_next_bin_hour	next bin datetime component 'hour'
golem_next_bin_second	next bin datetime component 'second'
golem_next_bin_microsecond	next bin datetime component 'microsecond'
golem_next_bin_iso	iso string repr for golem_next_bin_date
golem_next_bin_basic	iso basic string repr for golem_next_bin_date
golem_next_bin_silk	silk string repr for golem_next_bin_date
golem_view	the GolemView object which produced these tags
golem_inputs	a collected dictionary of all inputs that are defined in the tags
golem_outputs	a collected dictionary of all outputs that are defined in the tags

For intervals, time bins are bounded by golem_bin_date and golem_end_date. Spans, on the other hand, are bounded by golem_start_date and golem_end_date. For details on how intervals and spans relate to one another and format precisions, see *Intervals and Spans Explained*.

In addition to the standard golem tags defined above, all other tags, loops, inputs, and outputs defined in the initial

script configuration are available for use in templates. For example, assume a script makes the following declarations:

```
script.add_tag('in_types',  'in,inweb')
script.add_tag('out_types', 'out,outweb')
script.add_tag('month', "%(golem_month)02d/%(golem_year)d")
script.add_sensor_loop()
script.add_flow_tag('in_flow',  flow_type='in_types')
script.add_flow_tag('out_flow', flow_type='out_types')
script.add_output_template('juicy_set', ext='.set',
    description="Target 'juicy' set to generate.",
    mime_type='application/x-silk-ipset')
```

During each iteration of the processing loop, the tags dictionary will now include the following additional template entries:

in_types

```
"in,inweb"
```

out_types

```
"out,outweb"
```

month

```
"%(golem_month)02d/%(golem_year)d" \
    % (tags['golem_month'], tags['golem_year'])
```

`sensor`

the current iteration value of `script.get_sensors()`

in_flow

```
Flow_params(
    start_date = tags['golem_start_date'],
    end_date   = tags['golem_end_date'],
    sensors    = tags['sensor'],
    flow_type  = tags['in_types'])
```

out_flow

```
Flow_params(
    start_date = tags['golem_start_date'],
    end_date   = tags['golem_end_date'],
    sensors    = tags['sensor'],
    flow_type  = tags['out_types'])
```

juicy_set

```
"%(golem_name)s/%(sensor)s/"                              \
"%(golem_name)s.%(sensor)s.%(golem_start_date_iso)s.set" \
    % (tags['golem_name'], tags['sensor'],                \
       tags['golem_start_date_iso'])
```

Sometimes, depending on how loops and dependencies are arranged and how views are being manipulated, an input or output tag for the current view might contain multiple values (e.g. multiple filenames). In these cases, the resolved values are bundled into a `GolemArgs` object, which in turn resolves as a string of paths separated by spaces in the final command template.

2.10 Intervals and Spans Explained

Intervals and spans represent two different concepts.

An interval is a *processing interval* which represents how frequently the script is intended to produce results. This is roughly analogous to how frequently the script might be invoked via a cron job, except that golem scripts will back-fill missing results upon request and ignore intervals that appear to already have results present. An interval is always represented by the *first* timestamp contained within the interval.

A span, on the other hand, is a *data window* which represents how much input data the script is expected to consume, whether it be from a SiLK repository, results from other golem scripts, or other sources.

By default, the span of a golem script is the same as its interval. Not much surprising happens when the values are equal. They can be different, however. For example, a script might have a weekly interval yet consume 4 weeks worth of data for each of those weeks. Alternatively, a script might run every 4 weeks yet consume only a day's worth of data, akin to a monthly snapshot.

Intervals are *anchored* relative to a particular epoch. Intervals are always relative to midnight of the first Monday after January 1st, 1970 which was January 5th. Weeks therefore begin with Monday and multiples of weeks are always relative to that particular Monday. If another day of the week is desired, use the *set_skew* configuration function.

Spans are always anchored relative to the *end* of the processing interval.

In the tags dictionary provided for each processing loop, the interval is represented by the `golem_bin_date` entry and the span is represented by `golem_start_date` and `golem_end_date`. Given a 3 week interval and a 4 week span, for example, these values are aligned like so:

```
interval:              bin_date              next_bin_date
                          |                        |
                    |-------|-------|-------|-------|
                    |                                |
span:         start_date                          end_date
```

Note that `end_date` is not inclusive—its actual value is the value of `next_bin_date` minus one millisecond. `next_bin_date`, on the other hand, can be handy if you want to represent your results files by the end of the processing interval ("as of" `next_bin_date` as opposed to "begining with" `bin_date`).

Each of these entries are represented by `datetime.datetime` objects along with an assortment of formatted string representations. If both the interval and span have a magnitude of at least a day or more, the formatted string variations look like so:

Variation	Format
iso	`YYYY-MM-DD`
basic	`YYYYMMDD`
silk	`YYYY/MM/DD`

If either the interval or span is less than a day, hours are included:

Variation	Format
iso	`YYYY-MM-DDTHH`
basic	`YYYYMMDDTHH`
silk	`YYYY/MM/DDTHH`

In all of the examples covered in this documentation, result templates are all based on the `golem_bin_date` values, i.e. the processing interval. As the example diagram above illustrates, this may not be intuitive as to what data is represented in the results. It is up to script authors to decide how to name their results, but they should choose a convention, stay consistent with it, and document the decision.

2.11 Examples

2.11.1 Trivial Example

The following is a simple example using the Golem API that demonstrates basic templating. The script monitors an "incoming" directory for daily text files containing IP addresses and converts them into rwset files. A line by line explanation follows after the script:

```
#!/usr/bin/env python

from netsa.script.golem import script
from netsa.util import shell

script.set_title('Daily IP Sets')
script.set_description("""
    Convert daily IP lists into rwset files.
""")
script.set_version("0.1")
script.set_contact("H.P. Fnord <fnord@example.com>")
script.set_authors(["H.P. Fnord <fnord@example.com>"])

script.set_name('daily_set')
script.set_interval(days=1)
script.set_span(days=1)

script.add_golem_params()

script.set_repository('dat')

script.add_input_template('daily_txt',
    "/data/incoming/daily.%(golem_bin_iso)s.txt",
    description="Daily IP text files",
    mime_type='text/plain')

script.add_output_template('daily_set',
    "daily/daily.%(golem_bin_iso)s.set",
    description="Daily IP Sets",
    mime_type='application/x-silk-ipset')

def main():
    cmd = "rwsetbuild %(daily_txt)s %(daily_set)s"
    for tags in script.process():
        shell.run_parallel(cmd, vars=tags)

script.execute(main)
```

Here is the breakdown, line by line:

```
from netsa.script.golem import script
from netsa.util import shell
```

The first two lines import golem itself as well as `netsa.util.shell`, which assists in constructing command line templates and executing the resulting system commands and pipelines.

Next are some lines for configuring meta information about the script:

```
script.set_title('Daily IP Sets')
script.set_description("""
```

```
    Convert daily IP lists into rwset files.
""")
script.set_version("0.1")
script.set_contact("H.P. Fnord <fnord@example.com>")
script.set_authors(["H.P. Fnord <fnord@example.com>"])
```

Setting the title, description, and other meta-data of the script is the same as with the regular `netsa.script` *meta-data functions*.

Now for the golem-specific configuration:

```
script.set_name('daily_set')
```

Though optional, every golem script should have a short name, suitable for inclusion withing directory paths and filenames. It will be made available for use in templates as the `%(golem_name)s` tag. For groups of related scripts, the `set_suite_name` function is also available.

Next, the script must be told the size of its processing intervals and the size of its data window:

```
script.set_interval(days=1)
script.set_span(days=1)
```

These two parameters, *interval* and *span*, are the core configuration parameters for any golem script. The *interval* represents 'how often' this script is expected to generate results. Typically this would correspond to the schedule by which the script is invoked via a cron job. The *span* represents how far back the script will look for input data. The interval and span do not have to match as they do here—for example, a script might have a 'daily' interval which processes one week of data for each of those days.

Next, the script author will almost always want to enable the standard golem command line parameters:

```
script.add_golem_params()
```

There are three general categories of parameters (*basic*, *repository-related*, and *query-related*) which can be separately enabled; the line above enables all of these.

Next, the script can be told where its results will live:

```
script.set_repository('dat')
```

This line defines the location of the scripts output data repository. Note that some scripts can be designed for query purposes only and will therefore not need to define a repository location.

If the path provided is a relative path, it is assumed to be relative to the script's *home* path. See the `set_default_home` function for details on how the home path is configured or determined.

The script then defines a template for its input data:

```
script.add_input_template('daily_txt',
    "/data/incoming/daily.%(golem_bin_iso)s.txt",
    description="Daily IP text files",
    mime_type='text/plain')
```

This template assumes that the incoming files will correspond to the standard ISO-formatted datetime ('YYYY-MM-DD').

Next, an output template is defined:

```
script.add_output_template('daily_set',
    "daily/daily.%(golem_bin_iso)s.set",
    description="Daily IP Sets",
    mime_type='application/x-silk-ipset')
```

For each day of processing, a single rwset file will be generated. Once again, the standard ISO-formatted date is chosen for the template.

In both the input and output templates, the script uses the tag `%(golem_bin_iso)s`. This tag is an implicit template tag, automatically available for use within golem scripts. Other timestamps are also available, including portions of each timestamp (such as year, month, and day) for constructing more elaborate templates. For more details about the rest of these 'implicit' template tags, see *Usage of Tags, Loops, and Templates*.

Now for the actual processing loop:

```
def main():
    cmd = "rwsetbuild %(daily_txt)s %(daily_set)s"
    for tags in script.process():
        shell.run_parallel(cmd, vars=tags)

script.execute(main)
```

All 'real work' in a golem script should take place in a `main` function, which is subsequently passed to the `execute` function. In order for golem scripts to work properly, this must always be the case. Using the `netsa.script.execute` function instead will not work for a golem script.

The main entry point for looping across template values is the `process` function. This construct does a number of things, including creating output paths and checking for the existence of required inputs. On each iteration, a dictionary of template tags with resolved values is provided.

Every golem script has an intrinsic loop over processing intervals. In this example, our processing interval is once a day. If, via the command line parameters *`--first-date`* and *`--last-date`*, a window of 1 week had been specified, it would result in seven main iterations with the tag `%(golem_bin_iso)s` corresponding to a string representation of the beginning timestamp of each daily interval covered in the requested range. Unless told otherwise, a golem script will skip iterations which have already generated results.

2.11.2 Basic Example

The *Trivial Example* defined an input template that described daily text files without any explanation about where or how those files were produced. The following example assumes that the resulting set of addresses represent "observed internal hosts" and illustrates how the daily set files might be produced directly from queries to a SiLK repository. In order to do so, the script relies on a couple of SiLK command line tools:

```
#!/usr/bin/env python

from netsa.script.golem import script
from netsa.util import shell
from netsa import files

script.set_title('SiLK Daily Active Internal Hosts')
script.set_description("""
    Daily inventory of observed internal host activity.
""")
script.set_version("0.1")
script.set_contact("H.P. Fnord <fnord@example.com>")
script.set_authors(["H.P. Fnord <fnord@example.com>"])

script.set_name('daily_set')
script.set_interval(days=1)
script.set_span(days=1)

script.add_golem_params()
```

```
script.set_repository('dat')

script.add_tag('in_types',  'in,inweb')
script.add_tag('out_types', 'out,outweb')

script.add_output_template('internal_set',
    "internal/daily/daily.%(golem_bin_iso)s.set",
    description="Daily Internal host activity",
    mime_type='application/x-silk-ipset')

def main():
    for tags in script.process():
        tags['out_fifo'] = files.get_temp_pipe_name()
        tags['in_fifo']  = files.get_temp_pipe_name()
        cmd1 = [
            "rwfilter --start-date=%(golem_start_silk)s"
                " --end-date=%(golem_end_silk)s"
                " --type=%(in_types)s"
                " --proto=0-255 --pass=stdout",
            "rwset --sip=%(out_fifo)s"]
        cmd2 = [
            "rwfilter --start-date=%(golem_start_silk)s"
                " --end-date=%(golem_end_silk)s"
                " --type=%(out_types)s"
                " --proto=0-255 --pass=stdout",
            "rwset --dip=%(in_fifo)s"]
        cmd3 = [
            "rwsettool --union --output-path=%(internal_set)s"
                " %(out_fifo)s %(in_fifo)s"]
        shell.run_parallel(cmd1, cmd2, cmd3, vars=tags)

script.execute(main)
```

There are a couple of new techniques to note with this script. Below the standard meta-configuration are the following lines:

```
script.add_tag('in_types',  'in,inweb')
script.add_tag('out_types', 'out,outweb')
```

These two statements add a couple of simple template tags. All templates will now have access to the tags `%(in_types)s` and `%(out_types)s`, which will resolve to the strings `'in,inweb'` and `'out,outweb'`, respectively. This is equivalent to manually adding these entries to the `tags` dictionary down in the processing loop; predefining them here is a matter of style preference.

Next comes the main processing loop, which illustrates some more advanced usage of the `netsa.util.shell` module:

```
def main():
    for tags in script.process():
        tags['out_fifo'] = files.get_temp_pipe_name()
        tags['in_fifo']  = files.get_temp_pipe_name()
```

As mentioned earlier, a `tags` dictionary is provided for each processing interval and sensor. In the two lines within the processing loop, some additional tags are added to the dictionary. The lines illustrate how the `netsa.files` module can be used to create temporary named pipes so that data can be fed from one command to another.

These new template additions are then used in the construction of some command templates used to pull data from the SiLK repository:

```
cmd1 = [
    "rwfilter --start-date=%(silk_start)s"
        " --end-date=%(silk_end)s"
        " --type=%(in_types)s"
        " --proto=0-255 --pass=stdout",
    "rwset --sip=%(out_fifo)s"]
cmd2 = [
    "rwfilter --start-date=%(silk_start)s"
        " --end-date=%(silk_end)s"
        " --type=%(out_types)s"
        " --proto=0-255 --pass=stdout",
    "rwset --dip=%(in_fifo)s"]
cmd3 = [
    "rwsettool --union --output-path=%(internal_set)s"
        " %(out_fifo)s %(in_fifo)s"]
shell.run_parallel(cmd1, cmd2, cmd3, vars=tags)
```

The first two command templates utilize the template definitions defined earlier, `%(in_types)s` and `%(out_types)s`, along with the date ranges associated with each processing loop. Each of these commands sends its results into its respective named pipe. Finally, the third command uses `rwsettool` to create a union from the output of these named pipes and creates the rwset file defined by the output template. All three commands are run in parallel using the facilities of the `netsa.util.shell` module.

2.11.3 Basic Golem Dependency Example

The *Basic Example* provides a golem script that produces daily rwset files produced from queries to a SiLK repository. What if a weekly, rather than daily, summary of IP addresses is desired? One option would be to adjust the processing interval and span of the script, thereby pulling an entire week's worth of data from SiLK in the calls to `rwfilter`. An alternative is to utilize the daily sets from the original script as inputs and construct a weekly summary via the union of the daily sets for the week in question.

One of the core features of golem scripts is that they can be assigned as inputs to one other. Details such as how often the inputs are produced, the naming scheme, and synchronization across time bins is sorted out automatically by the golem scripts involved. Assume that the script in the *Basic Example* is called `daily_set.py`. The following example illustrates how to configure the dependency on this external script:

```
#!/usr/bin/env python

from netsa.script.golem import script
from netsa.util import shell

script.set_title('Weekly Active Internal Host Set')
script.set_description("""
    Aggregate daily internal activity sets over the last week.
""")
script.set_version("0.1")
script.set_contact("H.P. Fnord <fnord@example.com>")
script.set_authors(["H.P. Fnord <fnord@example.com>"])

script.set_name('weekly_set')
script.set_interval(weeks=1)
script.set_span(weeks=1)

script.add_golem_params()

script.set_repository('dat')
```

```
script.add_golem_input('daily_set.py', 'daily_set', cover=True)

script.add_output_template('weekly_set',
    "weekly/weekly.%(golem_bin_iso)s.set",
    description="Aggregated weekly sets.",
    mime_type='application/x-silk-ipset')

def main():
    cmd = "rwsettool --union --output-path=%(weekly_set)s %(daily_set)s"
    for tags in script.process():
        shell.run_parallel(cmd, vars=tags)

script.execute(main)
```

The first thing to note is that this script has a different interval and span:

```
script.set_interval(weeks=1)
script.set_span(weeks=1)
```

The script will produce weekly results and will expect to consume a week of data while doing so.

The input template is now defined as a dependency on the external script like so:

```
script.add_golem_input('daily_set.py', 'daily_set', cover=True)
```

The first argument is the name of the external script. For details on how relative paths to scripts are resolved, see the `add_golem_source` function.

The second argument is the output as defined within that external script. Golem scripts can have multiple outputs, so the specific output desired must be explicitly defined.

The third argument, the *cover* parameter, controls how this external output is synchronized across local processing intervals—in this case the processing interval of 1 week will be 'covered' by 7 days worth of inputs.

After the configuration of the weekly output template comes the main processing loop:

```
def main():
    cmd = "rwsettool --union --output-path=%(weekly_set)s %(daily_set)s"
    for tags in script.process():
        shell.run_parallel(cmd, vars=tags)

script.execute(main)
```

The output tag `%(weekly_set)s` is based on the `%(golem_bin_iso)s` timestamp, which in this case ends up being the date of the first Monday of each week in question. The `%(daily_set)s` tags represents 7 days of results—this will resolve to 7 individual filenames separated by whitespace in the eventual call to `rwsettool`.

2.11.4 Loop, Interval and Span Example

Golem scripts can define additional loops addition to the intrinsic loop over processing intervals. The following script is a modification of the script in the *Basic Example* which builds daily inventories by directly querying the SiLK repository. Rather than construct a monolithic inventory across all sensors, this version will construct inventories on a per-sensor basis by defining a template loop over a list of sensor names. Finally, it will illustrate the difference between intervals and spans by using a less frequent interval and a larger data window:

```
#!/usr/bin/env python

from netsa.script.golem import script
from netsa.util import shell
```

```
from netsa import files

script.set_title('Active Internal Hosts')
script.set_description("""
    Per-sensor inventory of observed internal host activity over a
    four week window of observation, generated every three weeks.
""")
script.set_version("0.1")
script.set_contact("H.P. Fnord <fnord@example.com>")
script.set_authors(["H.P. Fnord <fnord@example.com>"])

script.set_name('internal')
script.set_interval(weeks=3)
script.set_span(weeks=4)

script.add_golem_params()

script.set_repository('dat')

script.add_tag('in_types',  'in,inweb')
script.add_tag('out_types', 'out,outweb')

script.add_loop('sensor', ["S0", "S1", "S2", "S3"])

script.add_output_template('internal_set',
    "internal/internal.%(sensor)s.%(golem_bin_iso)s.set",
    description="Internal host activity",
    mime_type='application/x-silk-ipset')

def main():
    for tags in script.process():
        tags['out_fifo'] = files.get_temp_pipe_name()
        tags['in_fifo']  = files.get_temp_pipe_name()
        cmd1 = [
            "rwfilter --start-date=%(golem_start_silk)s"
                " --end-date=%(golem_end_silk)s"
                " --type=%(in_types)s"
                " --sensors=%(sensor)s"
                " --proto=0-255 --pass=stdout",
            "rwset --sip=%(out_fifo)s"]
        cmd2 = [
            "rwfilter --start-date=%(golem_start_silk)s"
                " --end-date=%(golem_end_silk)s"
                " --type=%(out_types)s"
                " --sensors=%(sensor)s"
                " --proto=0-255 --pass=stdout",
            "rwset --dip=%(in_fifo)s"]
        cmd3 = [
            "rwsettool --union --output-path=%(internal_set)s"
                " %(out_fifo)s %(in_fifo)s"]
        shell.run_parallel(cmd1, cmd2, cmd3, vars=tags)

script.execute(main)
```

The first thing to note is the new interval and span definitions:

```
script.set_interval(weeks=3)
script.set_span(weeks=4)
```

The script will produce results every 3 weeks and will expect to consume 4 weeks of data while doing so. This is the first example in which the interval and span are not equal. For more detail on the implications of this see *Intervals and Spans Explained*.

Further down in the script is the new loop definition:

```
script.add_loop('sensor', ["S0", "S1", "S2", "S3"])
```

With the addition of this line, for each 3-week processing interval, the script will return a separate `tags` dictionary for each sensor, setting the value of the `sensor` entry accordingly. Logically speaking this is equivalent to having two embedded 'for' loops, one for intervals and one for sensors.

This newly defined `%(sensor)s` tag is then used in the modified definition of the output template:

```
script.add_output_template('internal_set',
    "internal/internal.%(sensor)s.%(golem_bin_iso)s.set",
    description="Internal host activity",
    mime_type='application/x-silk-ipset')
```

Next comes the main processing loop. Note that it is *identical* to the processing loop in the earlier incarnation of the script. The interval and span were changed, an extra loop was introduced, and the output template was modified, but the essential processing logic remains unchanged.

2.11.5 SiLK Integration Example

The Golem API and *NetSA Scripting Framework* include a number of convenience functions and classes for interacting with a SiLK repository. The *Loop, Interval and Span Example* can be simplified using a few of these features as illustrated below:

```
#!/usr/bin/env python

from netsa.script.golem import script
from netsa.util import shell
from netsa import files

script.set_title('Active Internal Hosts')
script.set_description("""
    Per-sensor inventory of observed internal host activity over a
    four week window of observation, generated every three weeks.
""")
script.set_version("0.2")
script.set_contact("H.P. Fnord <fnord@example.com>")
script.set_authors(["H.P. Fnord <fnord@example.com>"])

script.set_name('internal')
script.set_interval(weeks=3)
script.set_span(weeks=4)

script.add_golem_params()

script.set_repository('dat')

script.add_tag('in_types',  'in,inweb')
script.add_tag('out_types', 'out,outweb')

script.add_sensor_loop()

script.add_flow_tag('in_flow',  flow_type='in_types')
```

```
script.add_flow_tag('out_flow', flow_type='out_types')

script.add_output_template('internal_set',
    "internal/internal.%(sensor)s.%(golem_bin_iso)s.set",
    description="Internal host activity",
    mime_type='application/x-silk-ipset')

def main():
    for tags in script.process():
        tags['out_fifo'] = files.get_temp_pipe_name()
        tags['in_fifo']  = files.get_temp_pipe_name()
        cmd1 = [
            "rwfilter %(out_flow)s --proto=0-255 --pass=stdout",
            "rwset --sip=%(out_fifo)s"]
        cmd2 = [
            "rwfilter %(in_flow)s --proto=0-255 --pass=stdout",
            "rwset --dip=%(in_fifo)s"]
        cmd3 = [
            "rwsettool --union --output-path=%(internal_set)s"
                " %(out_fifo)s %(in_fifo)s"]
        shell.run_parallel(cmd1, cmd2, cmd3, vars=tags)

script.execute(main)
```

The first difference to note is that rather than manually defining a loop over sensors, the following shorthand is used:

```
script.add_sensor_loop()
```

This line sets up a loop on the template tag `sensor` as before, but the list of sensors is automatically determined from the SiLK repository itself (see the `mapsid` command). The script also remembers that this particular loop involves sensors.

The next modification to note is the definition of two special SiLK-related compound tags:

```
script.add_flow_tag('in_flow',  flow_type='in_types')
script.add_flow_tag('out_flow', flow_type='out_types')
```

These statements create template entries bound to `netsa.script.Flow_params` objects which serve to simplify the construction of `rwfilter` command line templates.

Each call to `add_flow_tag` implicitly binds the `start_date` and `end_date` object attributes to the value of the template tags `golem_start_silk` and `golem_end_silk`. Given that a sensor-specific loop was declared earlier, the function calls will also bind the `sensors` attribute to the value of the `sensor` tag for each loop.

Additional tags can be bound to `netsa.script.Flow_params` attributes using keyword arguments. In this example, the `in_types` and `out_types` tags defined earlier in the script are bound to the `flow_type` attribute of each object.

The rest of the script proceeds as before, except that in the processing loop the `rwfilter` command templates are far more compact:

```
cmd1 = [
    "rwfilter %(out_flow)s --proto=0-255 --pass=stdout",
    "rwset --sip=%(out_fifo)s"]
cmd2 = [
    "rwfilter %(in_flow)s --proto=0-255 --pass=stdout",
    "rwset --dip=%(in_fifo)s"]
```

The `%(out_flow)s` and `%(in_flow)s` tags will each expand into four parameters in the eventual command string.

2.11.6 Synchronization Example

The following example will build a daily inventory of internal addresses that exhibit activity on source port 25. In order to limit the pool of addresses under consideration, it will utilize the internal inventory results generated by the *SiLK Integration Example*. Furthermore, it will utilize some additional SiLK-related tools in order to organize results into 'sensor groups' rather than under individual sensors. The following assumes that the prior inventory script is called `internal.py`:

```
#!/usr/bin/env python

from netsa.script.golem import script
from netsa.util import shell
from netsa import files

script.set_title('Daily Internal Port 25 Activity')
script.set_description("""
    Daily per-sensor-group inventory of observed internal host
    activity on port 25.
""")
script.set_version("0.1")
script.set_contact("H.P. Fnord <fnord@example.com>")
script.set_authors(["H.P. Fnord <fnord@example.com>"])

script.set_name('p25')
script.set_interval(days=1)
script.set_span(days=1)

script.add_golem_params()

script.set_repository('dat')

script.add_sensor_loop(auto_group=True)

script.add_tag('out_type', 'out,outweb')
script.add_flow_tag('out_flow', flow_type='out_type')

script.add_golem_input('internal.py', 'internal_set', join_on='sensor')

script.add_output_template('p25_set',
    "p25/p25.%(sensor_group)s.%(golem_bin_iso)s.set",
    description="Daily IP set for internal port 25 activity",
    mime_type='application/x-silk-ipset')

def main():
    for tags in script.process():
        tags['in_fifo'] = files.get_temp_pipe_name()
        cmd1 = [
            "rwsettool --union"
                " --output-path=%(in_fifo)s
                " %(internal_set)s"]
        cmd2 = [
            "rwfilter %(out_flow)s"
                " --proto=6"
                " --sport=25"
                " --packets=2-"
                " --sipset=%(in_fifo)s"
                " --pass=stdout",
            "rwset --sip-set=%(p25_set)s"]
```

```
        shell.run_parallel(cmd1, cmd2, vars=tags)

script.execute(main)
```

First, the script is configured to generate once per day using a span of one day:

```
script.set_interval(days=1)
script.set_span(days=1)
```

Next, the sensor loop is configured:

```
script.add_sensor_loop(auto_group=True)
```

This invocation of `add_sensor_loop` uses a new named parameter, *auto_group*, which loops over *groups* of related sensors rather than individual sensors. Normally, a single template tag `sensor` is added. When grouping is enabled for a sensor loop another tag `sensor_group` is added in addition to the `sensor` tag. So, for example, if there are three sensors in a group labeled 'LAB0', 'LAB1', and 'LAB2', these two template tags would expand into strings like so:

Tag	Value
`%(sensor)s`	`LAB0,LAB1,LAB2`
`%(sensor_group)s`	`LAB`

See the `add_loop` function for the details of how the above features work for generic, non-sensor-related, loops.

Next the script sets up the input dependency from the script in the *Basic Example* called `internal.py`:

```
script.add_golem_input('internal.py', 'internal_set', join_on='sensor')
```

When golem scripts use other golem script results as inputs, they are automatically synchronized across processing intervals. The basic rule is to synchronize on the latest external interval containing an end-point less than or equal to the end-point of the local interval under consideration.

The synchronization of any loops other than intervals must be explicitly configured. In this case, the *join_on* parameter is used to indicate that the external sensor loop and local sensor loop should align on each value of the `sensor` tag. This synchronization happens per-sensor and does not affect the eventual sensor grouping behavior.

Next, the output template is defined. Note the use of the `sensor_group` tag rather than `sensor`:

```
script.add_output_template('p25_set',
    "p25/p25.%(sensor_group)s.%(golem_bin_iso)s.set",
    description="Daily IP set for internal port 25 activity",
    mime_type='application/x-silk-ipset')
```

Followed by the processing loop:

```
def main():
    for tags in script.process():
        tags['in_fifo'] = files.get_temp_pipe_name()
        cmd1 = [
            "rwsettool --union"
                " --output-path=%(in_fifo)s
                " %(internal_set)s"]
        cmd2 = [
            "rwfilter %(out_flow)s"
                " --proto=6"
                " --sport=25"
                " --packets=2-"
                " --sipset=%(in_fifo)s"
                " --pass=stdout",
            "rwset --sip-set=%(p25_set)s"]
```

```
        shell.run_parallel(cmd1, cmd2, vars=tags)

script.execute(main)
```

Since sensors are being grouped, the `%(internal_set)s` tag for each loop potentially represents multiple input files, one for each individual sensor. The first command defines a template for `rwsettool` that sends a union of these per-sensor sets into the named pipe. The second command pipeline uses this merged set to filter the initial flows being examined by the `rwfilter` query.

When invoked on a regular basis, this script will produce a daily subset of the most recent per-sensor-group inventory for those internal IP addresses that have exhibited activity on source port 25.

2.11.7 Self Dependency Example

The *Synchronization Example* demonstrates how to configure an input dependency on the results of another golem script. It is also possible to configure dependencies on a golem script's *own* past results.

Recall that the *SiLK Integration Example* is configured with a 3-week interval and 4-week span. The 3-week interval was chosen due to the resource-intensive query across 4 weeks of data. Whereas this does produce internal inventories, the information is potentially less accurate over time (particularly during the final few days of the 3-week processing interval).

The inventory script can be modified to consume its own outputs and produce *delta encoded* results on a daily basis:

```
#!/usr/bin/env python

from netsa.script.golem import script
from netsa.util import shell
from netsa import files

script.set_title('Active Internal Hosts')
script.set_description("""
    Daily per-sensor inventory of observed internal host activity,
    delta-encoded using the prior four weeks of results.
""")
script.set_version("0.3")
script.set_contact("H.P. Fnord <fnord@example.com>")
script.set_authors(["H.P. Fnord <fnord@example.com>"])

script.set_name('internal')
script.set_interval(days=1)
script.set_span(days=1)

script.add_golem_params()

script.set_repository('dat')

script.add_tag('in_types',  'in,inweb')
script.add_tag('out_types', 'out,outweb')

script.add_sensor_loop()

script.add_flow_tag('in_flow',  flow_type='in_types')
script.add_flow_tag('out_flow', flow_type='out_types')

script.add_output_template('internal_set',
    "internal/internal.delta.%(sensor)s.%(golem_bin_iso)s.set",
    description="Delta set of internal host activity",
```

```
    mime_type='application/x-silk-ipset',
    scope=28)

script.add_self_input('prior_set', 'internal_set', offset=0)

def main():
    for tags in script.process():
        tags['out_fifo'] = files.get_temp_pipe_name()
        tags['in_fifo']  = files.get_temp_pipe_name()
        cmds = []
        cmds.append([
            "rwfilter %(out_flow)s --proto=0-255 --pass=stdout",
            "rwset --sip=%(out_fifo)s"])
        cmds.append([
            "rwfilter %(in_flow)s --proto=0-255 --pass=stdout",
            "rwset --dip=%(in_fifo)s"])
        cmds.append([
            "rwsettool --union --output-path=%(current_set)s"
                " %(out_fifo)s %(in_fifo)s"])
        if tags['prior_set']:
            tags['current_set'] = files.get_temp_pipe_name()
            cmds.append([
              "rwsettool --difference"
                  " --output-path=%(internal_set)s"
                  " %(current_out)s %(prior_set)s"])
        else:
            tags['current_set'] = tags['internal_set']

        shell.run_parallel(vars=tags, *cmds)

script.execute(main)
```

The goal is to generate a viable internal inventory on a daily basis with minimal overhead. The naive approach would be to define and interval of 1 day and leave the span as 4 weeks. This would pull 4 weeks of data *every single day* and construct a full inventory for that day. This is inefficient in terms of processing and storage. Instead, this script introduces a new concept called *scope*. Scope is used to indicate situations where a single interval of processing does not represent a complete analysis result.

First, the basics are configured:

```
script.set_interval(days=1)
script.set_span(days=1)
```

The script produces a daily result and expects to consume a single day's worth of 'regular' data while doing so. Next, the script must define its daily output template:

```
script.add_output_template('internal_set',
    "internal/internal.delta.%(sensor)s.%(golem_bin_iso)s.set",
    description="Delta set of internal host activity",
    mime_type='application/x-silk-ipset',
    scope=28)
```

This declaration shows the use of the new `scope` parameter. The scope indicates the number of processing interval outputs required to represent a *complete* result. Here, the scope is defined as 28 intervals (days in this case).

Now when other golem scripts use this script output as an input dependency, they will see 4 weeks of files relative to each day of interest. This also applies in cases where a golem script asks *itself* for prior results. An example of this is shown next:

```
script.add_self_input('prior_set', 'internal_set', offset=0)
```

This self-referential input dependency maps `internal_set` to a new template tag called `prior_set`.

By default, self-referential inputs have an offset of `-1` which excludes the results for the current processing interval. In cases such as this, where the goal is delta-encoding, the offset should be `0`. (The daily result being generated for the current day represents addresses not present in the last 27 days).

Next is the main processing loop:

```
def main():
    for tags in script.process():
        tags['out_fifo'] = files.get_temp_pipe_name()
        tags['in_fifo']  = files.get_temp_pipe_name()
        cmds = []
        cmds.append([
            "rwfilter %(out_flow)s --proto=0-255 --pass=stdout",
            "rwset --sip=%(out_fifo)s"])
        cmds.append([
            "rwfilter %(in_flow)s --proto=0-255 --pass=stdout",
            "rwset --dip=%(in_fifo)s"])
        cmds.append([
            "rwsettool --union --output-path=%(current_set)s"
                " %(out_fifo)s %(in_fifo)s"])
        if tags['prior_set']:
            tags['current_set'] = files.get_temp_pipe_name()
            cmds.append([
              "rwsettool --difference"
                  " --output-path=%(internal_set)s"
                  " %(current_out)s %(prior_set)s"])
        else:
            tags['current_set'] = tags['internal_set']

        shell.run_parallel(vars=tags, *cmds)

script.execute(main)
```

The core logic is similar to the earlier version. A new template tag, `current_set` is added to the `tags` dictionary for each iteration. Depending on circumstances, the value of this tag is set to one of two things: If no prior results are available, a regular rwset is constructed just as it was before. If prior results are available, however, the difference is taken between the current day's results and the union of up to 27 days of prior results.

This technique allows the reconstruction of an accurate 4-week internal inventory, for any particular day, by taking the union over the 28 days ending on that day.

Having made these changes, what now needs to be changed in the script from the *Synchronization Example* which depends on these internal sets as input?

Not a single thing.

The script in the *Synchronization Example* is already performing a union with `rwsettool` on the tag `%(internal_set)s` in order to merge data across sensors into sensor groups. Due to the `scope` declaration, the `%(internal_set)s` tag will now also include paths to the files for each of the 28 days required to reconstruct results.

2.12 Classes

2.12.1 GolemView

class netsa.script.golem.**GolemView**(*golem : Golem*[, *first_date : datetime*, *last_date : datetime*])

A GolemView object encapsulates a golem script model and is used to view and manipulate it in various ways. These different views are primarily accessed through the loop, outputs, and inputs methods.

Optional keyword arguments:

last_date The interval containing this datetime object is the last to be considered for processing. (default: most recent)

first_date The interval containing this datetime object is the first to be considered for processing. (default: *last_date*)

golem
The golem script model which this view manipulates.

first_bin
A datetime object representing the first processing interval for this view, as determined by the *first_date* and *last_date* parameters during construction. Defaults to last_bin.

last_bin
A datetime object representing the last processing interval for this view, as determined by the *last_date* and *first_date* parameters during construction. Defaults to the 'most recent' interval that does not overlap into the future, taking into account lag.

start_date
A datetime object representing the beginning of the first data span covered by this view. Spans can be larger (or smaller) than the defined interval, so this value is not necessarily equal to first_bin.

end_date
A datetime object representing the end of the last data span covered by this view. If the span is less than or equal to the interval, this is equal to last_bin + interval - 1 microsecond, otherwise it is equal to last_bin + span - 1 microsecond.

using([*golem : Golem*, *first_date : datetime*, *last_date : datetime*])
Return a copy of this GolemView object, optionally using new values for the following keyword arguments:

golem Use a different golem script model.

first_date Select a different starting time bin based on the provided datetime object.

last_date Select a different ending time bin based on the provided datetime object.

bin_dates() → datetime iter
Provide an iterator over datetime objects representing all processing intervals represented by this view.

bins() → GolemView iter
Provide an iterator over GolemView objects for each interval represented by this view.

group_by(*key : str*[, ...]) → (str tuple, GolemView) iter
Returns an iterator that yields a tuple with a primary key and GolemView object grouped by the provided keys. Each primary key is a tuple containing the current values of the keys provided to group_by. Iterating over the provided view objects will resolve any remaining loops if any remain that were not used for the provided key.

by_key(*key : str*) → (str, GolemView) iter
Similar to group_by but takes only a single key as an argument. Returns and iterator that yields view

objects for each value of the key; iterating over the provided view objects will resolve any remaining loops, if present.

product() → GolemView iter
Fully resolve the loops defined within this view. The 'outer' loop is always over intervals, followed by any other loops in the order in which they were defined. Each view thus provided is therefore fully resolved, with no loops remaining.

bin_count() → int
Return the number of intervals represented by this view, as defined by `first_bin` and `last_bin`.

loop_count() → int
Return the number of non-interval iterations represented by this view that are produced by resolving any defined loops.

sync_to(*other : GolemView [, count : int, offset : int, cover=False*) → GolemView
Given another `GolemView` object, return a version of *self* that has been synchronized to the given view object.

Optional keyword arguments:

count Synchronize to this many intervals of the given object (default: 1)

offset Synchronize to this many interval offsets behind the given object (default: 0)

cover Calculate a *count* necessary to cover all intervals represented by the given object (overrides *count* and *offset*)

trail Force the *end_date* of the new view to always be less than or equal to the *end_date* of the given view.

loop() → GolemTags
Return a `GolemTags` object representing this view.

outputs() → GolemOutputs
Return a `GolemOutputs` object representing this view.

inputs() → GolemInputs
Return a `GolemInputs` object representing this view.

__len__() → int
Return the number of fully-resolved iterations represented by this view, over intervals as well as any defined loops.

__iter__() → GolemView iter
Iterates over the views returned by the `product` method.

2.12.2 GolemTags

class netsa.script.golem.**GolemTags**(*golem : Golem*[, *first_date : datetime*, *last_date : datetime*])
Bases: `netsa.script.golem.GolemView`

A `GolemTags` object is used to examine resolved template tags produced by looping over intervals and other defined loops.

As well as the methods and attributes of `GolemView`, the following additional and overridden methods are available:

tags() → dict
Return a dictionary of resolved template tags for the current view (flattens tags across the loops that would result from invoking the `product` method).

__iter__() → dict iter

2.12.3 GolemOutputs

class netsa.script.golem.**GolemOutputs**(*golem : Golem*[, *first_date : datetime*, *last_date : datetime*])

Bases: netsa.script.golem.GolemView

A GolemOutputs object is used to examine resolved output templates, either for a specific iteration or aggregated across multiple iterations.

As well as the methods and attributes of GolemView, the following additional and overridden methods are available:

expand() → GolemArgs
Returns a GolemArgs object representing all resolved output templates for the current view.

__len__() → int
Return the number of resolved output templates for the current view.

__iter__() → str iter
Iterate over each resolved output template for the current view.

2.12.4 GolemInputs

class netsa.script.golem.**GolemInputs**(*golem : Golem*[, *first_date : datetime*, *last_date : datetime*])

Bases: netsa.script.golem.GolemView

A GolemInputs object is used to examine resolved input templates, either for a specific iteration or aggregated across multiple iterations.

As well as the methods and attributes of GolemView, the following additional and overridden methods are available:

expand() → GolemArgs
Returns a GolemArgs object representing all resolved input templates for the current view.

members() → GolemOutputs iter
Iterate over each golem script that provides inputs for this golem script, returning each as a synchronized GolemOutputs object.

__len__() → int
Return the number of resolved input templates for the current view.

__iter__() → str iter
Iterate over each resolved input template for the current view.

2.12.5 GolemArgs

class netsa.script.golem.**GolemArgs**(*item : str or str iter*[, ...])

A GolemArgs object encapsulates a list of resolved input or output templates destined to be used as a parameter in a tags dictionary. The constructor takes any number of strings, or string iterators, and flattens them into a unique sorted tuple. Individual items can be accessed and iterated over like a tuple. One keyword argument is accepted, *sep*, which will be used to join the items when rendered as a string. It defaults to a single space.

If a space is the separator, the object will resolve to a string of space-separated values and will properly resolve when passed to the `netsa.util.shell` module for command and pipeline execution.

Note that some file-related python functions (such as `open`) will complain if passed a single-value `GolemArgs` object (representing a single file name) without having first explicitly converted it to a string via `str` or index 0.

The length of a `GolemArgs` object represents the number of items it contains. These can be accessed via an index like a list. Two objects can be added and subtracted from one another, as with sets.

2.12.6 GolemProcess

class `netsa.script.golem.`**`GolemProcess`**(*gview : GolemView*[, *overwrite_outputs=False*, *skip_complete=True*, *keep_empty_outputs=False*, *skip_missing_inputs=False*, *optional_inputs : dict*])

A utility class for performing system-level interactions (such as checking for required inputs, pre-existing outputs, creating output paths, etc) while iterating over the provided view.

exception_handler Function for processing GolemException events. Takes the exception and the current GolemView as arguments.

overwrite_outputs Delete existing outputs prior to processing. (default: `False`)

keep_empty_outputs Consider zero-byte output results to be valid, otherwise they will be ignored or deleted when encountered, regardless of the value of *overwrite_results*. (default: `False`)

Most methods and attributes available from the `GolemTags` class are available through this class as well, with some behavioral changes as noted below. The following methods are in addition to those available from `GolemTags`:

`is_complete`() → bool
Returns a boolean value indicating whether processing has been completed for the intervals represented by this view.

`status`(*label : str*) → (str, bool) iter
Iterate over items within the given tag, returning a tuple containing the item string and its current status. Status is typically the size in bytes of each input or output, or `None` if it does not exist.

The following methods have slightly different behavior than that of `GolemTags`:

`using`([*gview : GolemView*, *overwrite_outputs=False*, *skip_complete=True*, *keep_empty_outputs=False*, *skip_missing_inputs=False*, *create_slots=True*, *optional_inputs : dict*])
Return a copy of this `GolemProcess` object, possibly replacing certain attributes corresponding to the keyword arguments in the constructor.

`product`() → GolemProcess iter
Return a `GolemProcess` object for each iteration over the processing intervals and loops defined for this process view, possibly performing system level tasks along the way (such as creating output paths and performing input checks). Iterations where processing is complete will be skipped, unless `overwrite_outputs` has been enabled for this object.

`bins`() → GolemProcess iter
Provide an iterator over `GolemProcess` objects for each processing interval represented by this view Iterations for which processing is complete will be skipped, unless `overwrite_outputs` has been enabled for this object.

`group_by`(*key : str*[, ...]) → (str tuple, GolemProcess) iter
Returns an iterator that yields a tuple with a primary key and a `GolemProcess` object, grouped by the provided keys. Each primary key is a tuple containing the current values of the keys provided. Iterating

over the resulting process objects process objects will resolve any remaining loops remaining in that view, if any. Views for which processing is complete will be skipped, unless `overwrite_outputs` has been enabled for this object.

by_key (*str*) → GolemProcess iter
Similar to `group_by` but takes a single key as an argument. Returns and iterator that yields `GolemProcess` objects for each value of the key. Views for which processing is complete will be skipped, unless `overwrite_outputs` has been enabled for this object.

__iter__ () → dict iter
Iterate over the views produced by the `product` method, yielding a dictionary of fully resolved template tags. Iterations for which processing is complete will be skipped, unless `overwrite_outputs` has been enabled for this object.

CHAPTER

THREE

NETSA.SQL — SQL DATABASE ACCESS

3.1 Overview

The normal flow of code that works with databases using the `netsa.sql` API looks like this:

```
from netsa.sql import *

select_stuff = db_query("""
    select a, b, c
      from test_table
      where a + b <= :threshold
    limit 10
""")

conn = db_connect("nsql-sqlite:/var/tmp/test_db.sqlite")

for (a, b, c) in conn.execute(select_stuff, threshold=5):
    print ("a: %d, b: %d, c: %d, a + b: %d" % (a, b, c, a+b))

# Alternatively:
for (a, b, c) in select_stuff(conn, threshold=5):
    print ("a: %d, b: %d, c: %d, a + b: %d" % (a, b, c, a+b))
```

First, the required queries are created as instances of the `db_query` class. Some developers prefer to have a separate module containing all of the queries grouped together. Others prefer to keep the queries close to where they are used.

When the database is to be used, a connection is opened using `db_connect`. The query is executed using `db_connection.execute`, or by calling the query directly. The result of that call is then iterated over and the data processed.

Connections and result sets are automatically closed when garbage collected. If you need to make sure that they are collected as early as possible, make sure the values are not kept around in the environment (for example, by assigning `None` to the variable containing them when your work is complete, if the variable won't be leaving scope for a while.)

3.2 Exceptions

exception `netsa.sql.`**`sql_exception`**(*message : str*)

Specific exceptions generated by `netsa.sql` derive from this.

exception netsa.sql.**sql_no_driver_exception**(*message : str*)
This exception is raised when no driver is installed that can handle a URL opened via db_connect.

exception netsa.sql.**sql_invalid_uri_exception**(*message : str*)
This exception is raised when the URI passed to db_connect cannot be parsed.

3.3 Connecting

netsa.sql.**db_connect**(*uri*[, *user : str*, *password : str*]) → db_connection
Given a database URI and an optional *user* and *password*, attempts to connect to the specified database and return a db_connection subclass instance.

If a user and password are given in this call as well as in the URI, the values given in this call override the values given in the URI.

Database URIs have the form:

```
<scheme>://<user>:<password>@hostname:port/<path>;<param>=<value>;...?<query>#<fragment>
```

Various pieces can be left out in various ways. Typically, the following form is used for databases with network addresses:

```
<scheme>://[user[:password]@]hostname[:port]/<dbname>[;<parameters>]
```

While the following form is used for databases without network addresses, or sometimes for connections to databases on the local host:

```
<scheme>:<dbname>[;user=<user>][;password=<password>][;<params>]
```

The user and password may always be given either in the network location or in the params. Values given in the db_connect call override either of those, and values given in the network location take priority over those given in the params.

Refer to a specific database driver for details on what URI scheme to use, and what other params or URI pieces may be meaningful.

3.4 Connections and Result Sets

class netsa.sql.**db_connection**(*driver : db_driver*, *variants : str list*)
An open database connection, returned by db_connect.

get_driver() → db_driver
Returns the db_driver used to open this connection.

clone() → db_connection
Returns a fresh open db_connection open to the same database with the same options as this connection.

execute(*query_or_sql : db_query or str*[, *<param_name>=<param_value>*, ...]) → db_result
Executes the given SQL query (either a SQL string or a query compiled with db_query) with the provided variable bindings for side effects. Returns a db_result result set if the query returns a result set, an int with the number of rows affected if available, or None otherwise.

commit()
Commits the current database transaction in progress. Note that if a db_connection closes without commit being called, the transaction will automatically be rolled back.

rollback()
Rolls back the current database transaction in progress. Note that if a db_connection closes without commit being called, the transaction will automatically be rolled back.

get_variants() → str seq
Returns which variant tags are associated with this connection.

class netsa.sql.**db_result**(*connection : db_connection*, *query : db_query*, *params : dict*)
A database result set, which may be iterated over.

get_connection() → db_connection
Returns the db_connection which produced this result set.

get_query() → db_query
Returns the db_query which was executed to produce this result set. (Note that if a string query is given to db_connection.execute, it will automatically be wrapped in a db_query, so this is always a db_query.)

get_params() → dict
Returns the dict of params which was given when this query was executed.

__iter__() → iter
Returns an iterator over the rows of this result set. Each row returned is a tuple with one item for each column. If there is only one column in the result set, a tuple of one column is returned. (e.g. (5,), not just 5 if there is a single column with the value five in it.)

It is an error to attempt to iterate over a result set more than once, or multiple times at once.

3.5 Compiled Queries

class netsa.sql.**db_query**(*sql : str*[, <*variant*> *: str*, ...])
A db_query represents a "compiled" database query, which will be used one or more times to make requests.

Whenever a query is executed using the db_connection.execute method, it may be provided as either a string or as a db_query object. If an object is used, it can represent a larger variety of possible behaviors. For example, it might give both a "default" SQL to run for the query, but also several specific versions meant to work with or around features of specific RDBMS products. For example:

```
test_query = db_query(
    """
        select * from blah
    """,
    postgres="""
        select * from pg_blah
    """,
    oracle="""
        select rownum, * from ora_blah
    """)
```

A db_query object is a callable object. If called on a connection, it will execute itself on that connection. Specifically:

```
test_query(conn, ...)
```

has the same effect as:

```
conn.execute(test_query, ...)
```

__call__ (*self*, *_conn : db_connection*[, <*param_name*>=<*param_value*>, ...]) → db_result
Execute this db_query on the given db_connection with parameters.

Note that the following methods are primarily of interest to driver implementors.

get_variant_sql (*accepted_variants : str seq*) → str
Given a list of accepted variant tags, returns the most appropriate SQL for this query. Specifically, this returns the first variant SQL given in the query which is acceptable, or the default SQL if none is acceptable.

get_variant_qmark_params (*accepted_variants : str seq*, *params : dict*) → str, seq
Like get_variant_format_parms, but for the DB API 2.0 'format' paramstyle (i.e. %s placeholders). This also escapes any percent signs originally present in the query.

get_variant_numeric_params (*accepted_variants : str seq*, *params : dict*) → str, seq
Like get_variant_format_params, but for the DB API 2.0 'numeric' paramstyle (i.e. :<n> placeholders).

get_variant_named_params (*accepted_variants : str seq*, *params : dict*) → str, dict
Like get_variant_format_params, but for the DB API 2.0 'named' paramstyle (i.e. :<name> placeholders). Note that this paramstyle is the native style required by the netsa.sql API.

get_variant_format_params (*accepted_variants : str seq*, *params : dict*) → str, seq
Converts the SQL and params of this query to a form appropriate for databases that use the DB API 2.0 'format' paramstyle (i.e. %s placeholders). Given a list of accepted variants and a dict of params, this returns the appropriate SQL with param placeholders converted to 'format' style, and a list of params suitable for filling those placeholders.

get_variant_pyformat_params (*accepted_variants : str seq*, *params : dict*) → str, dict
Like get_variant_format_params, but for the DB API 2.0 'pyformat' paramstyle (i.e. %(<name>)s placeholders). This also escapes any percent signs originally present in the query.

3.6 Implementing a New Driver

In order to implement a new database driver, you should create a new module that implements a subclass of db_driver, then calls register_driver with an instance of that subclass in order to register the new driver.

Your db_driver subclass will, of course, return subclasses of db_connection and db_result specific to your database as well. It should never be necessary to subclass db_query—that class is meant to be a database-neutral representation of a "compiled" query.

For most drivers, one of the get_variant_... methods of db_query should provide the query in a form that the underlying database can easily digest.

class netsa.sql.**db_driver**
A database driver, which holds the responsibility of deciding which database URLs it will attempt to open, and returning db_connection objects when a connection is successfully opened.

can_handle (*uri_scheme : str*) → bool
Returns True if this db_driver believes it can handle this database URI scheme.

connect (*uri : str*, *user: str or None*, *password : str or None*) → db_connection
Returns None if this db_driver cannot handle this database URI, or a db_connection subclass instance connected to the database if it can. The *user* and *password* parameters passed in via this call override any values from the URI.

netsa.sql.**register_driver** (*driver : db_driver*)
Registers a db_driver database driver object with the netsa.sql module. Driver modules generally register themselves, and this function is only of interest to driver writers.

netsa.sql.**unregister_driver**(*driver : db_driver*)
Removes a db_driver database driver object from the set of drivers registered with the netsa.sql module.

3.7 Experimental Connection Pooling

This version of netsa.sql contains experimental support for connection pooling. Connections in a pool will be created before they're needed and kept available for re-use. Note that since this API is still in the early stages of development, it is very likely to change between versions of *netsa-python*.

netsa.sql.**db_create_pool**(*uri*[, *user : str*, *password : str*], ...) → db_pool
Given a database URI, an optional *user* and *password*, and additional parameters, creates a driver-specific connection pool. Returns a db_pool from which connections can be obtained.

If a user and password (or other parameter) is given in this call as well as in the URI, the values given in this call override the values given in the URI.

See db_connect for details on database URIs.

class netsa.sql.**db_pool**
A pool of database connections for a single specific connection specification and pool configuration. See db_create_pool.

get_driver() → db_driver
Returns the db_driver used to open this connection.

connect() → db_connection
Returns a db_connection subclass instance from the pool, open on the database specified when the pool was created.

class netsa.sql.**db_driver**

create_pool(*uri*, *user : str or None*, *password : str or None*, ...) → db_pool
Returns None if this db_driver does not support pooled connections or cannot handle this database URI, or a db_pool subclass instance which can be used to obtain connections from a pool. The *user* and *password* parameters and any other parameters passed in via this call override any values from the URI.

3.8 Why Not DB API 2.0?

If you have experience with Python database APIs, you may be wondering why we have chosen to implement a new API rather than simply using the standard DB API 2.0.

In short, the problem is that the standard database API isn't really an API, but more a set of guidelines. For example, each database driver may use a different mechanism for providing query parameters. As another example, each API may also have different behaviors in the presence of threads.

Specifically, the sqlite module uses the 'pyformat' param style, which allows named parameters to queries which are passed as a dict, using Python-style formats. The sqlite3 module, on the other hand, uses the 'qmark' param style, where ? is used as a place-holder in queries, and the parameters are positional and passed in as a sequence.

We've done work to make sure that it's simple to implement netsa.sql-style drivers over the top of DB API 2.0-style drivers. In fact, all of the currently deployed drivers are of this variety. The only work that has to be done for such a driver is to start with one of the existing drivers, determine which paramstyle is being used, do any protection against threading issues that might be necessary, and turn the connection URI into a form that the driver you're using can handle.

Once that's done, you still have the issue that different databases may require different SQL to operate—but that's a lot easier to handle than "some databases use named parameters and some use positional". And, the variant system makes it easy to put different compatibility versions of the same query together.

CHAPTER

FOUR

NETSA.UTIL.SHELL — ROBUST SHELL PIPELINES

4.1 Overview

The `netsa.util.shell` module provides a facility for securely and efficiently running UNIX command pipelines from Python. To avoid text substitution attacks, it does not actually use the UNIX shell to process commands. In addition, it runs commands directly in a way that allows easier clean-up in the case of errors.

The following standard Python library functions provide similar capabilities, but without either sufficient text substitution protections or sufficient error-checking and recovery mechanisms:

- The `os.system` function
- The `subprocess` module
- The `popen2` module

Here are some examples, in increasing complexity, of the use of the `run_parallel` and `run_collect` functions:

Run a single process and wait for it to complete:

```
# Shell: rm -rf /tmp/test
run_parallel("rm -rf /tmp/test")
```

Start two processes and wait for both to complete:

```
# Shell: rm -rf /tmp/test1 & rm -rf /tmp/test2 & wait
run_parallel("rm -rf /tmp/test_dir_1",
             "rm -rf /tmp/test_dir_2")
```

Store the output of a command into a file:

```
# Shell: echo test > /tmp/testout
run_parallel(["echo test", ">/tmp/testout"])
```

Read the input of a command from a file (and put the ouput into another file):

```
# Shell: cat < /tmp/test > /tmp/testout
run_parallel(["</tmp/test", "cat", ">/tmp/testout"])
```

Append the output of a command to a file:

```
# Shell: echo test >> /tmp/testout
run_parallel(["echo test", ">>/tmp/testout"])
```

Pipe the output of one command into another command (and put the output into a file):

```
# Shell: echo test | sed 's/e/f/' > /tmp/testout
run_parallel(["echo test", "sed 's/e/f/'", ">/tmp/testout"])
```

Run two pipelines in parallel and wait for both to complete:

```
# Shell:
#    echo test | sed 's/e/f/' > /tmp/testout &
#    cat /etc/passwd | cut -f1 -d'|' > /tmp/testout2 &
#    wait
run_parallel(["echo test", "sed 's/e/f/'", ">/tmp/testout"],
             ["cat /etc/passwd", "cut -f1 -d'|'", ">/tmp/testout2"])
```

Run a single pipeline and collect the output and error output in the variables *out* and *err*:

```
# Shell: foo=`cat /etc/passwd | cut -f1 -d'|'`
(foo, foo_err) = run_collect("cat /etc/passwd", "cut -f1 -d'|'")
```

The following examples are more complicated, and require the use of the long forms of `command` and `pipeline` specifications. (All of the examples above have used the short-hand forms.) You should read the documentation for `command` and `pipeline` to see how the long forms and short-hand forms are related.

Run a pipeline, collect standard output of the pipeline to one file, and append standard error from all of the commands to another file:

```
# Shell: ( gen-data | cut -f1 -d'|' > /tmp/testout ) 2>> /tmp/testlog
run_parallel(pipeline("gen-data", "cut -f1 -d'|'", ">/tmp/testout",
                      stderr="/tmp/testlog", stderr_append=True))
```

Run a pipeline, collect standard output of the pipeline to one file, and collect standard error from one command to another file:

```
# Shell: ( gen-data 2> /tmp/testlog ) | cut -f1 -d'|' > /tmp/testout
run_parallel([command("gen-data", stderr="/tmp/testlog"),
              "cut -f1 -d'|'", ">/tmp/testout"])
```

Run a pipeline, collect standard output of the pipeline to a file, and ignore the potentially non-zero exit status of the `gen-data` command:

```
# Shell: (gen-data | cut -f1 -d'|' > /tmp/testout) || true
run_parallel([command("gen-data", ignore_exit_status=True),
              "cut -f1 -d'|'", ">/tmp/testout"])
```

Use long pipelines to process data using multiple named pipes:

```
# Shell:
#   mkfifo /tmp/fifo1
#   mkfifo /tmp/fifo2
#   tee /tmp/fifo1 < /etc/passwd | cut -f1 -d'|' | sort > /tmp/out1 &
#   tee /tmp/fifo2 < /tmp/fifo1 | cut -f2 -d'|' | sort > /tmp/out2 &
#   cut -f3 -d'|' < /tmp/fifo2 | sort | uniq -c > /tmp/out3 &
#   wait
run_parallel("mkfifo /tmp/fifo1",
             "mkfifo /tmp/fifo2")
run_parallel(
    ["</etc/passwd", "tee /tmp/fifo1", "cut -f1 -d'|'", ">/tmp/out1"],
    ["</tmp/fifo1", "tee /tmp/fifo2", "cut -f2 -d'|'", ">/tmp/out2"],
    ["</tmp/fifo2", "cut -f3 -d'|'", "sort", "uniq -c", ">/tmp/out3"])
```

4.2 Exceptions

exception netsa.util.shell.**PipelineException**(*message*, *exit_statuses*)

This exception represents a failure to process a pipeline in either run_parallel or run_collect. It can be triggered by any of the commands being run by the function failing (either because the file was not found or because the command's exit status was unacceptable.) The message contains a summary of the status of all of the sub-commands at the time the problem was discovered, including stderr output for each sub-command if available.

4.3 Building Commands and Pipelines

netsa.util.shell.**command**(<*command spec*>[, *stderr : str or file*, *stderr_append=False*, *ignore_exit_status=False*, *ignore_exit_statuses : int seq*]) → command

Interprets the arguments as a "command specification", and returns that specification as a value.

If there is only a single argument and it is a command, then a new command is returned with the options provided by this call. For example:

```
new_command = command(old_command, ignore_exit_status=True)
```

If there is only a single argument and it is a str, the string is parsed as if it were a simple shell command. (i.e. respecting single and double quotation marks, backslashes, etc.) For example:

```
# Shell: ls /etc
new_command = command("ls /etc")
```

If there is only a single argument and it is a list or a tuple, interpret it as being the argument vector for the command (with the first argument being the command to be executed.) For example:

```
# Shell: ls /etc
new_command = command(["ls", "/etc"])
```

If there are multiple arguments, each argument is taken as being one element of the argument vector, with the first bring the command to be executed. For example:

```
# Shell: ls /etc
new_command = command("ls", "/etc")
```

The following keyword arguments may be given as options to a command specification:

stderr Filename (str) or open file object of destination for stderr.

stderr_append True if *stderr* should be opened for append. Does nothing if *stderr* is already an open file.

ignore_exit_status If True, then the exit status for this command is completely ignored.

ignore_exit_statuses A list of numeric exit statuses that should not be considered errors when they are encountered.

In addition, these options may be "handed down" from the pipeline call, or from run_parallel or run_collect. If so, then options given locally to the command take precedence.

Example: Define a command spec using a single string:

```
# Shell: ls -lR /tmp/foo
c = command("ls -lR /tmp/foo")
```

Example: Define a command as the same as an old command with different options:

```
d = command(c, ignore_exit_status=True)
```

Example: Define a command using a list of strings:

```
# Shell: ls -lR /tmp/foo
e = command(["ls", "-lR", "/tmp/foo"])
```

Example: Define a command using individual string arguments:

```
# Shell: ls -lR /tmp/foo
f = command("ls", "-lR", "/tmp/foo")
```

Short-hand Form:

In the `pipeline`, `run_parallel`, and `run_collect` functions, commands may be given in a short-hand form where convenient. The short-hand form of a command is a single string. Here are some examples:

```
"ls -lR"              =>  command(["ls", "-lR"])
"echo test test a b"  =>  command(["echo", "test", "test", "a", "b"])
"echo 'test test' a"  =>  command(["echo", "test test", "a"])
"'weird program'"     =>  command(["weird program"])
```

There is no way to associate options with a short-hand `command`. If you wish to redirect error output or ignore exit statuses, you will need to use the long form.

Variable Expansion:

When commands are executed, variable expansion is performed. The expansions are provided by the argument *vars* to `run_parallel` or `run_collect`. Note that commands are split into arguments *before* this expansion occurs, which is a security measure. This means that no matter what whitespace or punctuation is in an expansion, it can't change the sense of the command. The down side of that is that on occasions when you would like to add multiple arguments to a command, you must construct the `command` using the list syntax.

Expansion variable references are placed using the Python String formatting operations.

Here is an example substitution, showing how `%(target)s` becomes a single argument before the subsitution occurs.

```
("ls -lR %(target)s", vars={'target': 'bl ah"'}) =>
("ls", "-lR", "%(target)s", vars={'target': 'bl ah"'}) =>
("ls", "-lR", 'bl ah "')
```

If the value to be substituted implements the method `get_argument_list`, which takes no arguments and returns a list of strings, then those strings are included as multiple separate arguments. This is an expert technique for extending commands at call-time for use internal to APIs.

```
("ls -lR %(targets)s", vars={'targets': special_container}) =>
("ls", "-lR", "target1", "target2", ...)
```

Functions as Commands:

In addition to executable programs, Python functions may also be used as commands. This is useful if you wish to do processing of data in a sub-process as part of a pipeline without needing to have auxilliary Python script files. However, this is an advanced technique and you should fully understand the subtleties before making use of it.

When a Python function is used as a command, the process will *fork* as normal in preparation for executing a new command. However, instead of *exec*-ing a new executable, the Python function is called. When the Python function completes (either successfully or unsuccessfully), the child process exits immediately.

If you intend to use this feature, be sure that you know how the lifecycles of various objects will behave when the Python interpreter is forked and two copies are running at once.

The command function is called with *vars* (as given to `run_parallel` or `run_collect`) as its first argument, and the remainder of *argv* from calling `command` as its remaining arguments.

netsa.util.shell.**pipeline**(<*pipeline spec*>[, *stdin : str or file*, *stdout : str or file*, *stdout_append=False*, ...]) → pipeline

Interprets the arguments as a "pipeline specification", and returns that specification as a value.

If there is only a single argument and it is a `pipeline`, then a new pipeline is returned with the options provided by this call. For example:

```
new_pipeline = pipeline(old_pipeline, stdout="/tmp/newfile")
```

If there is only a single argument and it is a `list` or a `tuple`, interpret it as being a list of commands and I/O redirection short-hands to run in the pipeline. For example:

```
# Shell: ls /etc | sort -r
new_pipeline = pipeline(["ls /etc", "sort -r"])
```

If there are multiple arguments, these arguments are treated as a list of commands and I/O redirection short-hands (as if they were passed as a single list.) For example:

```
# Shell: ls /etc | sort -r
new_pipeline = pipeline("ls /etc", "sort -r")
```

The following keyword arguments may be given as options to a pipeline specification:

stdin Filename (`str`) or open `file` object of source for stdin.

stdout Filename (`str`) or open `file` object of destination for stdout.

stdout_append `True` if *stdout* should be opened for append. Does nothing if *stdout* is already an open file.

Because these options are so common, they may also be given in short-hand form. If the first command in the pipeline is a string starting with <, the remainder of the string is intepreted as a filename for stdin. If the last command in the pipeline is a string starting with > or >>, the remainder of the string is interpreted as a filename for stdout (and if >> was used, it is opened for append.)

In addition, any unrecognized keyword arguments will be provided as defaults for any `command` specifications used in this pipeline. (So, for example, if you give the *ignore_exit_status* option to `pipeline`, all of the commands in that pipeline will use the same value of *ignore_exit_status* unless they have their own overriding setting.)

Example: Define a pipeline using a list of commands:

```
# Shell: ls -lR /tmp/foo | sort > /tmp/testout
a = pipeline(command("ls -lR /tmp/foo"),
             command("sort"),
             stdout="/tmp/testout")
```

Example: Define the same pipeline using the short-hand form of commands, and the shorthand method of setting stdout:

```
# Shell: ls -lR /tmp/foo | sort > /tmp/testout
b = pipeline("ls -lR /tmp/foo",
             "sort",
             ">/tmp/testout")
```

Example: Define the same pipeline using a list instead of multiple arguments:

```
# Shell: ls -lR /tmp/foo | sort > /tmp/testout
c = pipeline(["ls -lR /tmp/foo",
              "sort",
              ">/tmp/testout"])
```

Example: Define a new pipeline which is the same as an old pipeline but with different options:

```
d = pipeline(c, stdout="/tmp/newout")
```

Short-hand Form:

In the `run_parallel` command, pipelines may be given in a short-hand form where convenient. The short-hand form of a pipeline is a list of commands and I/O redirection short-hands. Here are some examples:

```
# Shell: ls /tmp/die | xargs rm
["ls /tmp/die", "xargs rm"]  =>  pipeline(["ls /tmp/die", "xargs rm"])
# Shell: sort < /tmp/testin > /tmp/testsort
["</tmp/testin", "sort", ">/tmp/testsort"]  =>
         pipeline(["sort"], stdin="/tmp/testin", stdout="/tmp/testsort")
```

Note that although you can set *stdin*, *stdout*, and *stdout_append* using the short-hand form (by using the I/O redirection strings at the start and end of the list), you cannot set these options to open `file` objects, only to filenames. You also set other options to be passed down to the individual commands.

Variable Expansion:

As in `command`, pipelines have variable expansion. Most variable expansion happens inside the actual commands in the pipeline. However, variable expansion also occurs in filenames provided for the *stdin* and *stdout* options. For example:

```
# Shell: ls -lR > $output_file
pipeline("ls -lR", ">%(output_file)s")
pipeline("ls -lR", stdout="%(output_file)s")
```

4.4 Running Pipelines

`netsa.util.shell.`**`run_parallel`**`(`*<pipeline spec>*, ...[, *vars : dict*, ...])

Runs a series of commands (as specified by the arguments provided) by forking and establishing pipes between commands. Raises `PipelineException` and kills off all remaining subprocesses if any one command fails.

Each argument is passed to the `pipeline` function to create a new pipeline, which allows the short-hand form of pipelines (as `list` short-hands) to be used.

The following keyword arguments may be given as *options* to `run_parallel`:

vars A dictionary of variable substitutions to make in the `command` and `pipeline` specifications in this *run_parallel* call.

Additional keyword arguments will be passed down as default values to the `pipeline` and `command` specifications making up this `run_parallel` call.

The `run_parallel` function returns the list of exit codes of the processes in each pipeline as a list of lists. Each list corresponds to a pipeline, in the order in which they were passed into the function. Each element represents a process in the pipeline, in the order they were defined in the pipeline. If a process is not run (e.g., because a process preceding it in the pipeline fails), the exit status will be *None*.

Example: Run three mkdirs in parallel and fail if any of them fails:

```
# Shell: mkdir a & mkdir b & mkdir c & wait
run_parallel("mkdir a", "mkdir b", "mkdir c")
```

Example: Make a fifo, then afterwards, use it to do some work. (Try making a typo in here and watch it kill everything off instead of hanging forever.)

```
# Shell: mkfifo test.fifo
run_parallel("mkfifo test.fifo")
# Shell:
#   cat /etc/passwd | sort -r | cut -f1 -d: > $f &
#   cat $f | sed -e 's/a/b/g' > $f2 &
#   wait
run_parallel(["cat /etc/passwd", "sort -r", "cut -f1 -d:", ">%(f)s"],
             ["cat %(f)s", "sed -e 's/a/b/g'", ">%(f2)s"],
             vars={'f': 'test.fifo', 'f2': 'test.txt'})
```

Example: run two pipelines in parallel, then investigate their processes' exit statuses:

```
# Shell:
#   ls -l | grep ^d &
#   cat /etc/passwd | sort -r | cut -f1 -d: &
#   wait
exits = run_parallel(["ls -l", "grep ^d"],
                     ["cat /etc/passwd", "sort -r", "cut -f1 -d:"])
# If all complete successfully, exits will be:
#  [[0, 0], [0, 0, 0]]
```

netsa.util.shell.**run_collect**(<*command spec*>, ...[, *vars : dict*, ...]) → str, str

Runs a series of commands specifying a single pipeline by forking and establishing pipes between commands. The output of the final command is collected and returned in the result. stderr across all commands is returned in the result. The final result is a tuple (*stdout*, *stderr*)

Raises `PipelineException` and kills off all remaining subprocesses if any one command fails.

The arguments are passed as arguments to a single call of the `pipeline` function to create a pipeline specification. That is: each argument is a `command` specification. Note that this is not the same as `run_parallel`, which interprets its arguments as multiple `pipeline` specifications.

You can also redirect stderr independently for each command if needed, allowing you to send some stderr to `/dev/null` or another destination instead of collecting it.

Example: Reverse sort the output of `ls -1` and store the output and error in the variables *a_stdout* and *a_stderr*:

```
# Shell: ls -1 | sort -r
(a_stdout, a_stderr) = run_collect("ls -1", "sort -r")
```

Example: Do the same as the above, but run `ls -1` on a named directory instead of the current working directory:

```
# Shell: ls -1 $dir | sort -r
(b_stdout, b_stderr) = run_collect("ls -1 %(dir)s", "sort -r",
                                   vars={'dir': 'some_directory'})
```

Example: The following *does not collect output*, but instead writes it to a file. If there were any error output, it would be returned in the variable *c_stderr*:

```
# Shell: ls -1 | sort -r > test.out
(empty_stdout, c_stderr) = run_collect("ls -1", "sort -r", ">test.out")
```

netsa.util.shell.**run_collect_files**(<*command spec*>, ...[, *vars : dict*, ...]) → file, file

Runs a series of commands like `run_collect`, but returns open file objects for *stdout* and *stderr* instead of strings.

Example: Iterate over the lines of `ls -l | sort -r` and print them out with line numbers:

```
# Shell: ls -l | sort -r
(f_stdout, f_stderr) = run_collect_files("ls -l", "sort -r")
for (line_no, line) in enumerate(f_stdout):
    print ("%3d %s" % (line_no, line[:-1]))
```

CHAPTER

FIVE

NETSA_SILK — NETSA PYTHON/PYSILK SUPPORT

The netsa_silk module contains a shared API for working with common Internet data in both netsa-python and PySiLK. If netsa-python is installed but PySiLK is not, the less efficient but more portable pure-Python version of this functionality that is included in netsa-python is used. If PySiLK is installed, then the high-performance C version of this functionality that is included in PySiLK is used.

This document describes version 1.0 of the netsa_silk module API.

5.1 IPv6 Support

Depending on which version of the netsa_silk functionality is in use, IPv6 support may not be present or may be limited. The following functions allow determining what variety of IPv6 support is available.

netsa_silk.**has_IPv6Addr**() → bool
: Returns True if the most basic form of IPv6 support—support for IPv6 addresses—is available. If it is not available, then this function returns False, IPAddr will raise ValueError when given an IPv6 address, and any call to IPv6Addr will raise NotImplementedError.

See also ip_set.supports_ipv6.

5.2 IP Addresses

An IP address is represented by either an IPv4Addr or an IPv6Addr. Both of these are subclasses of the generic IPAddr class.

class netsa_silk.**IPAddr**(*address : str or IPAddr*) → IPv4Addr or IPv6Addr
: Converts the input into an IP address, either IPv4 or IPv6. Returns either an IPv4Addr or IPv6Addr object, depending on whether the given input is parsed as an IPv4 or an IPv6 address.

  If IPv6 address support is not available (has_IPv6Addr returns False), then attempting to parse an IPv6 address will raise ValueError.

  Examples:

```
>>> addr1 = IPAddr('192.160.1.1')
>>> addr2 = IPAddr('2001:db8::1428:57ab')
>>> addr3 = IPAddr('::ffff:12.34.56.78')
>>> addr4 = IPAddr(addr1)
>>> addr5 = IPAddr(addr2)
```

class netsa_silk.**IPv4Addr**(*address : int or str or IPAddr*)

Converts the input into a new IPv4 address. If the integer input is too large a value, if the string input is unparseable as an IPv4 address, or if the IPAddr input is not convertible to an IPv4 address, raises a ValueError.

IPv4Addr is a subclass of IPAddr.

Examples:

```
>>> addr1 = IPv4Addr('192.160.1.1')
>>> addr2 = IPv4Addr(IPAddr('::ffff.12.34.56.78'))
>>> addr3 = IPv4Addr(addr1)
>>> addr4 = IPv4Addr(0x10000000)
```

class netsa_silk.**IPv6Addr**(*address : int or str or IPAddr*)

Converts the input into a new IPv6 address. If the integer input is too large a value, or if the string input is unparseable as an IPv6 address, raises a ValueError. If the input is an IPv4Addr, the address is converted to IPv6 via IPv4-mapped address embedding. ("1.2.3.4" becomes "::ffff:1.2.3.4").

If IPv6 address support is not available (has_IPv6Addr returns False), then calling IPv6Addr will raise NotImplementedError.

IPv6Addr is a subclass of IPAddr.

Examples:

```
>>> addr1 = IPv6Addr('2001:db8::1428:57ab')
>>> addr2 = IPv6Addr(IPAddr('192.168.1.1'))
>>> addr3 = IPv6Addr(addr1)
>>> addr4 = IPv6Addr(0x10000000000000000000000)
```

5.2.1 Comparisons

Whenever an IPv4 address is compared to an IPv6 address, the IPv4 address is converted to IPv6 using IPv4-mapped address embedding. This means that IPAddr('0.0.0.0') equals IPAddr('::ffff:0.0.0.0'). You can distinguish IPv4 addresses from IPv6 address by using the is_ipv6() method.

Operation	Result
a == *b*	if *a* is equal to *b*, then True, else False
a != *b*	if *a* is equal to *b*, then False, else True
a < *b*	if *a*'s integer representation is less than *b*'s, then True, else False
a <= *b*	if *a*'s integer representation is less than or equal to *b*'s, then True, else False
a >= *b*	if *a*'s integer representation is greater than or equal to *b*'s, then True, else False
a > *b*	if *a*'s integer representation is greater than *b*'s, then True, else False

5.2.2 Conversions

The following operations and methods may be used to convert between IPv4 and IPv6 addresses and between IP addresses and other types.

Operation	Result	Notes
`addr.is_ipv6()`	if *addr* is an IPv6 address, then `True`, else `False`	
`addr.to_ipv4()`	the IPv4 equivalent of *addr*, or `None` if no such equivalent exists	(1)
`addr.to_ipv6()`	the IPv6 equivalent of *addr*	(2)
`int(addr)`	the integer representation of *addr*	(3)
`str(addr)`	the human-readable string representation of *addr*	(4)
`addr.padded()`	a zero-padded human-readable string representation of *addr*	(5)
`addr.octets()`	a tuple containing each octet of *addr* in network byte order as an unsigned integer	

Notes:

1. If the address is already an IPv4 address, does nothing. If the address is an IPv6 address using IPv4-mapped address embedding (e.g. "::ffff:1.2.3.4"), returns the equivalent IPv4 address. Otherwise, returns `None`.
2. If the address is already an IPv6 address, does nothing. If the address is an IPv4 address, returns the address converted to an IPv6 address using IPv4-mapped address embedding.
3. If the address is an IPv4 address, returns an unsigned 32-bit integer value. If the address is an IPv6 address, returns an unsigned 128-bit integer value.
4. The address is returned in its canonical form.
5. If the address is an IPv4 address, returns a string of the form "xxx.xxx.xxx.xxx", where each field is one octet of the address as a zero-padded base-10 integer. If the address is an IPv6 address, returns a string of the form "xxxx:xxxx:xxxx:xxxx:xxxx:xxxx:xxxx:xxxx", where each field is two octets of the address as a zero-padded base-16 integer.

5.2.3 Masking

Masking operations return a copy of the address with all non-masked bits set to zero.

Operation	Result	Notes
`addr.mask(mask)`	the address *addr* masked by the bits of the address *mask*	(1)
`addr.mask_prefix(len)`	the address *addr* masked to a length of *len* prefix bits	

Notes:

1. If *addr* is an IPv6 address but *mask* is an IPv4 address, *mask* is converted to IPv6 and then the mask is applied. If *addr* was not an IPv4-mapped embedded IPv6 address, the result may not be what was expected. Prefer `addr.mask_prefix(len)` for IPv6 addresses when possible.

5.3 IP Sets

While there are multiple different IP set implementations with different qualities, `netsa_silk` provides a standard API for these sets, and a standard mechanism for acquiring a value of some set when you don't need a specific implementation.

class `netsa_silk.`**`ip_set`**`(`[*iterable*]`)`

Returns a new IP set object from an unspecified implementation that matches the following API. The elements of the set must be IP addresses. The values in *iterable* may be `IPAddr` objects, strings parsable by `IPAddr`, `IPWildcard` objects, or strings parsable by `IPWildcard`.

In all of the following descriptions, *s*, *s1*, and *s2* must be `ip_set` obejcts, *addr* may be an `IPAddr` object or a string that is parsable as an `IPAddr`. *iterable* may be any iterable containing `IPAddr` objects, `IPWildcard` objects, and strings, as described above.

5.3.1 Cardinality

Since IP sets can grow very large, an additional method for querying the cardinality which supports large values is available.

Operation	Result	Notes
`s.cardinality()`	the cardinality of *s*	
`len(s)`	the cardinality of *s*	(1)

Notes:

1. If the cardinality of *s* is exceptionally large, this may raise `OverflowError` due to limitations in Python. Using `s.cardinality()` is highly preferred for IP sets.

5.3.2 Membership

Operation	Result
addr `in` *s*	if *addr* is a member of *s*, then‘‘True‘‘, else `False`
addr `not in` *s*	if *addr* is a member of *s*, then `False`, else `True`

5.3.3 Comparison

Operation	Result
s1 `==` *s2*	if *s* has exactly the same elements as *s2*, then `True`, else `False`
s1 `!=` *s2*	if *s1* does not have exactly the same elements as *s2*, then `True`, else `False`
`s.isdisjoint(`*iterable*`)`	if *s* has no elements in common with *iterable*, then `True`, else `False`
`s.issubset(`*iterable*`)`	if every element of *s* is also in *iterable*, then `True`, else `False`
s1 `<=` *s2*	if every element of *s1* is also in *s2*, then `True`, else `False`
s1 `<` *s2*	if *s1* `<` *s2* and *s1* `!=` *s2*, then `True`, else `False`
`s.issuperset(`*iterable*`)`	if every element of *iterable* is also in *s*, then `True`, else `False`
s1 `>=` *s2*	if every element of *s2* is also in *s1*, then `True`, else `False`
s1 `>` *s2*	if *s1* `>` *s2* and *s1* `!=` *s2*, then `True`, else `False`

5.3.4 Manipulation

The following operations return a new IP set with the desired changes, while leaving the original IP set unmodified.

Operation	Result
`s.union(`*iterable*`, ...)`	a set with all elements that are in *s* or any *iterable*
s1 `|` *s2*	a set with all elements that are in *s1* or *s2*
`s.intersection(`*iterable*`, ...)`	a set with only elements that are in *s* and every *iterable*
s1 `&` *s2*	a set with only elements that are in both *s1* and *s2*
`s.difference(`*iterable*`, ...)`	a set with all elements that are in *s* but in no *iterable*
s1 `-` *s2*	a set with all elements that are in *s1* but not in *s2*
`s.symmetric_difference(`*iterable*`)`	a set with all elements that are in *s* or *iterable* but not both
s1 `^` *s2*	a set with all elements that are in *s1* or *s2* but not both
`s.copy()`	a shallow copy of *s*

5.3.5 Modification

The following operations modify the target IP set in place.

Operation	Result	Notes
s.update(*iterable, ...*)	updates *s* by adding all elements from each *iterable*	
s1 |= *s2*	updates *s1* by adding all elements from *s2*	
s.intersection_update(*iterable, ...*)	updates *s* by removing all elements that do not appear in every *iterable*	
s1 &= *s2*	updates *s1* by removing all elements that do not appear in *s2*	
s.difference_update(*iterable, ...*)	updates *s* by removing all elements that appear in any *iterable*	
s1 -= *s2*	updates *s1* by removing all elements that appear in *s2*	
s.symmetric_difference_update(*iterable*)	updates *s*, keeping only elements found in *s* or *iterable* but not both	
s1 ^= *s2*	updates *s1*, keeping only elements found in *s1* or *s2* but not both	
s.add(*addr*)	adds *addr* to *s*	
s.remove(*addr*)	removes *addr* from *s*	(1)
s.discard(*addr*)	removes *addr* from *s*	
s.pop()	removes and returns an arbitrary element from *s*	(2)
s.clear()	removes all elements from *s*	

Notes:

1. Raises `KeyError` if *addr* is not in *s*.
2. Raises `KeyError` if *s* was empty.

5.3.6 CIDR Block Iteration

ip_set.**cidr_iter**() → (IPAddr, int) iter
: Returns an iterator over the CIDR blocks covered by this IP set. Each value in the iterator is a pair (*addr*, *prefix_len*) where *addr* is the first IP address in the block, and *prefix_len* is the prefix length of the block.

5.3.7 IPv6 Support

Some `ip_set` implementations do not provide IPv6 support. Such an implementation will raise an `exception.TypeError` on any attempt to add an `IPv6Addr` to the set. The following class method can be used to determine if a given implementation has IPv6 support:

classmethod ip_set.**supports_ipv6**() → bool
: Returns `True` if this IP set implementation provides support for IPv6 addresses, or `False` otherwise.

5.4 IP Wildcards

An `IPWildcard` object represents the specification of a set of IP addresses using SiLK IP wildcard syntax. Not all sets of IP addresses can be represented by a single IP wildcard.

class netsa_silk.**IPWildcard**(*wildcard : str or IPWildcard*)
: Returns a new `IPWildcard` object constructed from *wildcard*. The string *wildcard* may contain an IP address, an IP address with a CIDR prefix designation, an integer, an integer with a CIDR prefix designation, or a SiLK wildcard expression. In SiLK wildcard notation, a wildcard is represented as an IP address in canonical form with each octet (for IPv4 addresses) or octet pair (IPv6) holding a single value, a range of values, a comma-separated list of values and ranges, or the character 'x' to accept all values.

Examples:

```
>>> wild1 = IPWildcard('1.2.3.0/24')
>>> wild2 = IPWildcard('ff80::/16')
>>> wild3 = IPWildcard('1.2.3.4')
>>> wild4 = IPWildcard('::FFFF:0102:0304')
>>> wild5 = IPWildcard('16909056')
>>> wild6 = IPWildcard('16909056/24')
>>> wild7 = IPWildcard('1.2.3.x')
>>> wild8 = IPWildcard('1:2:3:4:5:6:7:x')
>>> wild9 = IPWildcard('1.2,3.4,5.6,7')
>>> wild10 = IPWildcard('1.2.3.0-255')
>>> wild11 = IPWildcard('::2-4')
>>> wild12 = IPWildcard('1-2:3-4:5-6:7-8:9-a:b-c:d-e:0-ffff')
```

5.4.1 Membership

The primary operation on `IPWildcard` objects is testing whether an address is contained in the set covered by the wildcard. Both `IPAddr` and `str` values may be tested for membership.

Operation	Results	Notes
addr `in` *wildcard*	*addr* matches *wildcard*, then `True`, else `False`	(1)
addr `not in` *wildcard*	*addr* matches *wildcard*, then `False`, else `True`	(1)

Notes:

1. *addr* may be an `IPAddr` or a string. Strings are automatically converted as with `IPAddr(`*addr*`)`

5.4.2 Other

The following additional operations are available on `IPWildcard` objects:

Operation	Result
`str(`*wildcard*`)`	the string that was used to construct *wildcard*
wildcard`.is_ipv6()`	if *wildcard* contains IPv6 addresses then `True`, else `False`

5.5 TCP Flags

A `TCPFlags` object represents the eight bits of flags from a TCP session.

class `netsa_silk.`**`TCPFlags`**(*value : int or str or TCPFlags*)

Returns a new `TCPFlags` object with the given flags set. If *value* is an integer, it is interpreted as the bitwise integer representation of the flags. If *value* is a string, it is interpreted as a case-insensitive sequence of letters indicating individual flags, and optional white space. The mapping is described below.

Each supported flag has an assigned letter in string representations, is available as an attribute on `TCPFlags` values, and is available as a `TCPFlags` constant in `netsa_silk`:

Flag	Meaning	Letter	TCPFlags attribute	netsa_silk constant
FIN	No more data from sender	F	*flags*.fin	TCP_FIN
SYN	Synchronize sequence numbers	S	*flags*.syn	TCP_SYN
RST	Reset the connection	R	*flags*.rst	TCP_RST
PSH	Push Function	P	*flags*.psh	TCP_PSH
ACK	Acknowledgment field significant	A	*flags*.ack	TCP_ACK
URG	Urgent Pointer field significant	U	*flags*.urg	TCP_URG
ECE	ECN-echo (RFC 3168)	E	*flags*.ece	TCP_ECE
CWR	Congestion window reduced (RFC 3168)	C	*flags*.cwr	TCP_CWR

5.5.1 Bit-Manipulation

The following bit-manipulation operations are available on TCPFlags objects:

Operation	Result
~*flags*	the bitwise inversion (not) of *flags*
flags1 & *flags2*	the bitwise intersection (and) of *flags1* and *flags2*
flags1 | *flags2*	the bitwise union (or) of *flags1* and *flags2*
flags1 ^ *flags2*	the bitwise exclusive disjunction (xor) of *flags1* and *flags2*

5.5.2 Conversions

The following operations and methods may be used to convert TCPFlags objects into other types.

Operation	Result
int(*flags*)	the integer value of the flags set in *flags*
str(*flags*)	a string representation of the flags set in *flags*
flags.padded()	a space-padded column aligned string representation of the flags set in *flags*
bool(*flags*)	if any flag is set in *flags*, then True, else False

5.5.3 Matching

TCPFlags.**matches**(*flagmask : str*) → bool

The TCPFlags.matches method may be used to determine if a TCPFlags value matches a given flag/mask specification. The specification is given as a string containing a set of flags that must be set, optionally followed by a slash and a set of flags that must be checked. (i.e. if "A" is not in the flag list but is in the mask, it must be false. If "U" is not in either, it may have any value.) For example, *flags*.matches('S/SA') would return True if SYN was set and ACK was not set in *flags*.

Examples:

```
>>> flags = TCPFlags('SAU')
>>> flags.matches('S')
True
>>> flags.matches('SA/SA')
True
>>> flags.matches('S/SP')
True
>>> flags.matches('S/SA')
False
>>> flags.matches('SP/SP')
False
>>> flags.matches('A/SA')
False
```

5.6 Support for SiLK versions before 3.0

Although `netsa_silk` is only fully supported by SiLK as of version 3.0, some legacy support for older versions of PySiLK is available. Some specific things to watch for if you need to work with older SiLK versions:

1. In `IPv6Addr` the `octets` method is not available. Other conversion operations still work, however.
2. For `TCPFlags`, the lower-case flag name attributes (e.g. *`flags`*`.syn`) on the object are not available. To work around this, use the `TCPFlags.matches` method, or perform bitwise operations on the constants in the module. (For example, instead of `if` *`flags`*`.fin:` *`stuff...`*, use `if` *`flags`* `&` `TCP_FIN:` *`stuff...`*.)

CHAPTER

SIX

DATA MANIPULATION

6.1 netsa.data.countries — Country and Region Codes

Definitions of country and region names and codes as defined by ISO 3166-1 and the UN Statistics Division. The information in this module is current as of September 2015.

netsa.data.countries.**get_area_numeric**(*code : int or str*) → int
Given a country or region code as one of the following:

- String containing ISO 3166-1 alpha-2 code
- String containing ISO 3166-1 alpha-3 code
- String or integer containing ISO 3166-1 numeric code
- String containing DNS top-level domain alpha-2 code
- String or integer containing UN Statistics Division numeric region code

Returns the appropriate ISO 3166-1 or UN Statistics Division numeric code as an integer.

Note that some regions and other special items that are not defined by ISO 3166-1 or the UN Statistics Division are encoded as ISO 3166-1 user-assigned code elements.

Raises KeyError if the code is unrecognized.

netsa.data.countries.**get_area_name**(*code : int or str*) → str
Given a country or region code as a string or integer, returns the name for the country or region.

Raises KeyError if the country or region code is unrecognized.

netsa.data.countries.**get_area_tlds**(*code : int or str*) → str list
Given a country or region code as a string or integer, returns a list of zero or more DNS top-level domains for that country or region.

Raises KeyError if the country or region code is unrecognized.

netsa.data.countries.**get_country_numeric**(*code : int or str*) → int
Given a country code as a string or integer, returns the ISO 3166-1 numeric code for the country.

Raises KeyError if the country code is unrecognized.

netsa.data.countries.**get_country_name**(*code : int or str*) → str
Given a country code as a string or integer, returns the name for the country.

Raises KeyError if the country code is unrecognized.

netsa.data.countries.**get_country_alpha2**(*code : int or str*) → str
Given a country code as a string or integer, returns the ISO 3166-1 alpha-2 code for the country, or `None` if that is not possible.

Raises `KeyError` if the country code is unrecognized.

netsa.data.countries.**get_country_alpha3**(*code : int or str*) → str
Given a country code as a string or integer, returns the ISO 3166-1 alpha-3 code for the country, or `None` if that is not possible.

Raises `KeyError` if the country code is unrecognized.

netsa.data.countries.**get_country_tlds**(*code : int or str*) → str list
Given a country code as a string or integer, returns a list of zero or more DNS top-level domains for that country.

Raises `KeyError` if the country code is unrecognized.

netsa.data.countries.**iter_countries**() → int iter
Returns an iterator which yields all known ISO 3166-1 numeric country codes as integers, including user-assigned code elements in use.

netsa.data.countries.**get_region_numeric**(*code : int or str*) → int
Given a UN Statistics Division region code as a string or integer, returns the code as an integer.

Raises `KeyError` if the region code is unrecognized.

netsa.data.countries.**get_region_name**(*code : int or str*) → str
Given a region code as a string or integer, returns the name for the region.

Raises `KeyError` if the region code is unrecognized.

netsa.data.countries.**get_region_tlds**(*code : int or str*) → str list
Given a region code as a string or integer, returns a list of zero or more DNS top-level domains for that region.

Raises `KeyError` if the region code is unrecognized.

netsa.data.countries.**iter_regions**() → int iter
Returns an iterator which yields all top-level UN Statistics Division numeric region codes as integers. This includes Africa, the Americas, Asia, Europe, Oceania, and Other.

netsa.data.countries.**iter_region_subregions**(*code : int or str*) → int iter
Given the code for a containing region, returns an iterator which yields all second-level UN Statistics Division numeric region codes as integers.

Raises `KeyError` if the region code is unrecognized.

netsa.data.countries.**iter_region_countries**(*code : int or str*) → int iter
Given the code for a containing region, returns an iterator which yields as integers all ISO 3166-1 numeric country codes that are part of that region.

6.2 `netsa.data.format` — Formatting Data for Output

The `netsa.data.format` module contains functions useful for formatting data to be displayed in human-readable output.

6.2.1 Numbers

netsa.data.format.**num_fixed**(*value : num*[, *units : str*, *dec_fig=2*, *thousands_sep : str*]) → str
Format *value* using a fixed number of figures after the decimal point. (e.g. "1234" is formatted as "1234.00")

If *units* is provided, this unit of measurement is included in the output. *dec_fig* specifies the number of figures after the decimal point.

If *thousands_sep* is given, it is used to separate each group of three digits to the left of the decimal point.

Examples:

```
>>> num_fixed(1234, 'm')
'1234.00m'
>>> num_fixed(1234, 'm', dec_fig=4)
'1234.0000m'
>>> num_fixed(1234.5678, 'm', dec_fig=0)
'1235m'
>>> num_fixed(123456789, dec_fig=3, thousands_sep=",")
'123,456,789.000'
```

netsa.data.format.**num_exponent**(*value : num*[, *units : str*, *sig_fig=3*]) → str

Format *value* using exponential notation. (i.e. "1234" becomes "1.23e+3" for three significant digits, or "1.234e+4" for four significant digits.) If *units* is provided, this unit of measurement is included in the output. *sig_fig* is the number of significant figures to display in the formatted result.

Examples:

```
>>> num_exponent(1234, 'm')
'1.23e+3m'
>>> num_exponent(1234, 'm', sig_fig=4)
'1.234e+3m'
>>> num_exponent(1234.5678, 'm', sig_fig=6)
'1.23457e+3m'
>>> num_exponent(123456789, sig_fig=2)
'1.2e+8'
>>> num_exponent(123456, sig_fig=6)
'1.23456e+5'
```

netsa.data.format.**num_prefix**(*value : num*[, *units : str*, *sig_fig=3*, *use_binary=False*, *thousands_sep : str*]) → str

Format *value* using SI prefix notation. (e.g. 1k is 1000) If *units* is provided, this unit of measurement is included in the output. *sig_fig* is the number of significant figures to display in the formatted result.

If *use_binary* is `True`, then SI binary prefixes are used (e.g. 1Ki is 1024). Note that there are no binary prefixes for negative exponents, so standard prefixes are always used for such cases.

For very large or very small values, exponential notation (e.g. "1e-30") is used.

If *thousands_sep* is given, it is used to separate each group of three digits to the left of the decimal point.

Examples:

```
>>> num_prefix(1024, 'b')
'1.02kb'
>>> num_prefix(1024, 'b', use_binary=True)
'1.00Kib'
>>> num_prefix(12345, 'b', sig_fig=2)
'12kb'
>>> num_prefix(12345, 'b', sig_fig=7)
'12345.00b'
>>> num_prefix(12345678901234567890, 'b')
'12.3Eb'
>>> num_prefix(12345678901234567890, 'b', sig_fig=7)
'12345.68Pb'
>>> num_prefix(1234567890123456789012345, 's')
'1.23e+24s'
```

```
>>> num_prefix(0.001, 's')
'1.00ms'
>>> num_prefix(0.001, 's', use_binary=True)
'1.00ms'
```

6.2.2 Dates and Times

Dates and times may be formatted to a variety of precisions. The formatting functions support the following precisions, except where otherwise noted: `DATETIME_YEAR`, `DATETIME_MONTH`, `DATETIME_DAY`, `DATETIME_HOUR`, `DATETIME_MINUTE`, `DATETIME_SECOND`, `DATETIME_MSEC`, and `DATETIME_USEC`.

`netsa.data.format.`**`datetime_silk`**(*value : datetime*[, *precision=DATETIME_SECOND*]) → str

Format *value* as a SiLK format date and time (`YYYY/MM/DDTHH:MM:SS.SSS`). Implicitly coerces the time into UTC.

precision is the amount of precision that should be included in the output.

For a more general way to round times, see `netsa.data.times.bin_datetime`. See also `datetime_silk_hour` and `datetime_silk_day` for the most common ways to format incomplete dates in SiLK format.

Examples:

```
>>> t = netsa.data.times.make_datetime("2010-02-03T04:05:06.007")
>>> datetime_silk(t)
'2010/02/03T04:05:06'
>>> datetime_silk(t, precision=DATETIME_YEAR)
'2010'
>>> datetime_silk(t, precision=DATETIME_MONTH)
'2010/02'
>>> datetime_silk(t, precision=DATETIME_DAY)
'2010/02/03'
>>> datetime_silk(t, precision=DATETIME_HOUR)
'2010/02/03T04'
>>> datetime_silk(t, precision=DATETIME_MINUTE)
'2010/02/03T04:05'
>>> datetime_silk(t, precision=DATETIME_SECOND)
'2010/02/03T04:05:06'
>>> datetime_silk(t, precision=DATETIME_MSEC)
'2010/02/03T04:05:06.007'
>>> datetime_silk(t, precision=DATETIME_USEC)
'2010/02/03T04:05:06.007000'
```

`netsa.data.format.`**`datetime_silk_hour`**(*value : datetime*) → str

Format *value* as a SiLK format datetime to the precision of an hour (`YYYY/MM/DDTHH`). Implicitly coerces time into UTC. This is shorthand for `datetime_silk(value, precision=DATETIME_HOUR)`.

Example:

```
>>> t = netsa.data.times.make_datetime("2010-02-03T04:05:06.007")
>>> datetime_silk_hour(t)
'2010/02/03T04'
```

`netsa.data.format.`**`datetime_silk_day`**(*v : datetime*) → str

Format *value* as a SiLK format datetime to the precision of a day (`YYYY/MM/DD`). Implicitly coerces time into UTC. This is shorthand for `datetime_silk(value, precision=DATETIME_DAY)`.

Example:

```
>>> t = netsa.data.times.make_datetime("2010-02-03T04:05:06.007")
>>> datetime_silk_day(t)
'2010/02/03'
```

netsa.data.format.**datetime_iso**(*value : datetime*[, *precision=DATETIME_SECOND*]) → str

Format *value* as an ISO 8601 extended format date and time (`YYYY/MM/DDTHH:MM:SS.SSSSSS[TZ]`). Includes timezone offset unless the value has no timezone or the value's timezone is UTC.

precision is the amount of precision that should be included in the output.

For a more general way to round times, see `netsa.data.times.bin_datetime`. See also `datetime_silk_hour` and `datetime_silk_day` for the most common ways to format incomplete dates in SiLK format.

Examples:

```
>>> t = netsa.data.times.make_datetime("2010-02-03T04:05:06.007008")
>>> datetime_iso(t)
'2010-02-03T04:05:06'
>>> datetime_iso(t, precision=DATETIME_YEAR)
'2010'
>>> datetime_iso(t, precision=DATETIME_MONTH)
'2010-02'
>>> datetime_iso(t, precision=DATETIME_DAY)
'2010-02-03'
>>> datetime_iso(t, precision=DATETIME_HOUR)
'2010-02-03T04'
>>> datetime_iso(t, precision=DATETIME_MINUTE)
'2010-02-03T04:05'
>>> datetime_iso(t, precision=DATETIME_SECOND)
'2010-02-03T04:05:06'
>>> datetime_iso(t, precision=DATETIME_MSEC)
'2010-02-03T04:05:06.007'
>>> datetime_iso(t, precision=DATETIME_USEC)
'2010-02-03T04:05:06.007008'
>>> t = netsa.data.times.make_datetime("2010-02-03T04:05:06.007008+09:10", utc_only=False)
>>> datetime_iso(t)
'2010-02-03T04:05:06+09:10'
```

netsa.data.format.**datetime_iso_day**(*value : datetime*) → str

Format *value* as an ISO 8601 extended format date to the precision of a day (`YYYY-MM-DD[TZ]`). Includes timezone offset unless the value has no timezone or the value's timezone is UTC. This is shorthand for `datetime_iso(value, precision=DATETIME_DAY)`.

Example:

```
>>> t = netsa.data.times.make_datetime("2010-02-03T04:05:06.007")
>>> datetime_iso_day(t)
'2010-02-03'
>>> t = netsa.data.times.make_datetime("2010-02-03T04:05:06.007+03:00", utc_only=False)
>>> datetime_iso_day(t)
'2010-02-03+03:00'
```

netsa.data.format.**datetime_iso_basic**(*value : datetime*[, *precision=DATETIME_SECOND*]) → str

Format *value* as an ISO 8601 basic (compact) format date and time (`YYYYMMDDTHHMMSS.SSSSSS[TZ]`). Includes timezone offset unless the value has no timezone or the value's timezone is UTC.

precision is the amount of precision that should be included in the output. Note that in accordance with the ISO 8601 specification, this format does not support the `DATETIME_MONTH` precision, because `YYYYMM` and

YYMMDD would be potentially ambiguous.

For a more general way to round times, see netsa.data.times.bin_datetime. See also datetime_silk_hour and datetime_silk_day for the most common ways to format incomplete dates in SiLK format.

Examples:

```
>>> t = netsa.data.times.make_datetime("2010-02-03T04:05:06.007008")
>>> datetime_iso_basic(t)
'20100203T040506'
>>> datetime_iso_basic(t, precision=DATETIME_YEAR)
'2010'
>>> datetime_iso_basic(t, precision=DATETIME_DAY)
'20100203'
>>> datetime_iso_basic(t, precision=DATETIME_HOUR)
'20100203T04'
>>> datetime_iso_basic(t, precision=DATETIME_MINUTE)
'20100203T0405'
>>> datetime_iso_basic(t, precision=DATETIME_SECOND)
'20100203T040506'
>>> datetime_iso_basic(t, precision=DATETIME_MSEC)
'20100203T040506.007'
>>> datetime_iso_basic(t, precision=DATETIME_USEC)
'20100203T040506.007008'
>>> t = netsa.data.times.make_datetime("2010-02-03T04:05:06.007008+09:10", utc_only=False)
>>> datetime_iso_basic(t)
'20100203T040506+0910'
```

netsa.data.format.**timedelta_iso**(*value : timedelta*) → str

Format a datetime.timedelta object as a str in ISO 8601 duration format, minus 'year' and 'month' designators (P[n]DT[n]H[n]M[n]S). Fractional seconds will represented using decimal notation in the seconds field.

Note that conversions between units are precise and do not take into account any calendrical context. In particular, a day is exactly 24*3600 seconds, just like datetime.timedelta uses.

If you apply the resulting timedelta to a datetime and the interval happens to include something like leap seconds adjust your expectations accordingly.

Since datetime.timedelta has no internal representation of months or years, these units are never included in the result.

Examples:

```
>>> t1 = netsa.data.times.make_datetime("2010-02-03T04:05:06.007008")
>>> t2 = netsa.data.times.make_datetime("2010-02-04T05:06:07.008009")
>>> t3 = netsa.data.times.make_datetime("2010-02-03T04:06:06.008009")
>>> d1 = t2 - t1
>>> d2 = t3 - t1
>>> timedelta_iso(d1)
>>> 'P1DT1H1M1.001001S'
>>> timedelta_iso(d2)
>>> 'PT1M0.001001S'
```

6.3 netsa.data.nice — "Nice" Numbers for Chart Bounds

A set of functions to produce ranges of aesthetically-pleasing numbers that have the specified length and include the specified range. Functions are provided for producing nice numeric and time-based ranges.

netsa.data.nice.**nice_ticks**(*lo : num*, *hi : num*[, *ticks=5*, *inside=False*]) → num, num, num iter
Find 'nice' places to put *ticks* tick marks for numeric data spanning from *lo* to *hi*. If *inside* is `True`, then the nice range will be contained within the input range. If *inside* is `False`, then the nice range will contain the input range. To find nice numbers for time data, use `nice_time_ticks`.

The result is a tuple containing the minimum value of the nice range, the maximum value of the nice range, and an iterator over the tick marks.

See also `nice_ticks_seq`.

netsa.data.nice.**nice_ticks_seq**(*lo : num*, *hi : num*[, *ticks=5*, *inside=False*]) → num seq
A convenience wrapper of `nice_ticks` to return the nice range as a sequence.

netsa.data.nice.**nice_time_ticks**(*lo : datetime*, *hi : datetime*[, *ticks=5*, *inside=False*, *as_datetime=True*]) → datetime/int, datetime/int, datetime/int iter
Find 'nice' places to put *ticks* tick marks for time data spanning from *lo* to *hi*. If *inside* is `True`, then the nice range will be contained within the input range. If *inside* is `False`, then the nice range will contain the input range. To find nice numbers for numerical data, use `nice_ticks`.

The result is a tuple containing the minimum value of the nice range, the maximum value of the nice range, and an iterator over the ticks marks. If *as_datetime* is `True`, the result values will be `datetime.datetime` objects. Otherwise, the result values will be numbers of seconds since UNIX epoch. Regardless, the return value is expressed in UTC.

See also `nice_time_ticks_seq`.

netsa.data.nice.**nice_time_ticks_seq**(*lo : datetime*, *hi : datetime*[, *ticks=5*, *inside=False*, *as_datetime=True*]) → datetime/int seq
A convenience wrapper of `nice_time_ticks` to return the nice range as a sequence.

6.4 `netsa.data.times` — Time and Date Manipulation

netsa.data.times.**make_datetime**(*v : num or str or datetime or mxDateTime*[, *utc_only=True*]) → datetime
Produces a `datetime.datetime` object from a number (seconds from UNIX epoch), a string (in ISO format, SiLK format, or old SiLK format), or a `datetime.datetime` object. If *utc_only* is `True`, coerces the result to be in the UTC time zone.

If the mxDateTime library is installed, this function also accepts `mxDateTime` objects.

netsa.data.times.**bin_datetime**(*dt : timedelta*, *t : datetime*[, *z=UNIX_EPOCH : datetime*]) → datetime
Returns a new `datetime.datetime` object which is the floor of the `datetime.datetime` *t* in a *dt*-sized bin. For example:

```
bin_datetime(timedelta(minutes=5), t)
```

will return the beginning of a five-minute bin containing the time *t*. If you have very specific requirements, you can replace the origin point for binning (*z*) with a time of your choice. By default, the UNIX epoch is used, which is appropriate for most uses.

netsa.data.times.**make_timedelta**(*v : timedelta or str*) → timedelta
Produces a `datetime.timedelta` object from a string (in ISO 8601 duration format) or a `datetime.timedelta` object.

Since `datetime.timedelta` objects do not internally support units larger than 'days', ISO 8601 strings containing month or year designations are discouraged. If these units are encountered in the string, however, they converted to days using a precise formula. This is an exact conversion that does not take into account any

calendrical context. If you apply the result to a datetime, and the interval happens to include leapseconds, or if you expect to land on the same day of the month while adding 'months' or 'years', adjust your expectations accordingly.

Example of ISO 8601: 'P1DT1H1M1S'

This translates as a period of '1 day' with time offset of '1 hour, 1 minute, and 1 second'. Fields are optional, the 'P' is required, as is the 'T' if using any units smaller than a day. A zero-valued timedelta can be represented as 'P0D'.

netsa.data.times.**divmod_timedelta**(*n : timedelta*, *d : timedelta*) → int, timedelta
Given two datetime.timedelta objects, return the number of times the second one (denominator) fits into the first one (numerator), along with any remainder expressed as another timedelta.

6.4.1 Date Snappers

class netsa.data.times.**DateSnapper**(*size : timedelta*[, *epoch=UNIX_EPOCH : datetime*])
Class for date bin manipulations

date_aligned(*date*) → bool
Tests whether or not the provided date is the beginning datetime.datetime for the containing time bin.

See make_datetime for more detail on acceptable formats for date descriptors.

date_bin(*date*) → datetime
Returns a datetime.datetime object representing the beginning of the date bin containing the provided date ('snapping' the date into place)

See make_datetime for more detail on acceptable formats for date descriptors.

date_bin_end(*date*) → datetime
Returns a datetime.datetime object representing the last date of the date bin which contains the provided date.

See make_datetime for more detail on acceptable formats for date descriptors.

date_binner(*dates : date seq*) → seq
Given a list of datetimes, returns an iterator which produces tuples containing two datetime objects for each provided datetime. The first value of the tuple is the beginning of the date bin containing the datetime in question and the second value is the original datetime.

See make_datetime for more detail on acceptable formats for datetime descriptors.

date_clumper(*date_ranges : seq*) → datetime seq
Given a list of date ranges, return a list of date bins that intersect the union of the given date ranges. Each date range in the provided list can be a single datetime descriptor or a tuple representing a beginning and end datetime for the range.

See make_datetime for more detail on acceptable formats for date descriptors.

date_sequencer(*date_list : date seq*) → seq
Given a list of datetimes, returns an iterator which produces tuples containing two datetime objects. The first value of the tuple is the begining of the date bin and the second value is the original datetime. Both bins and datetimes will be repeated where necessary to fill in gaps not present in the original list of datetimes.

If, for example, the span between each successive datetime in the provided list is smaller than the defined bin size, the same bin will be returned for each datetime residing in that bin. (An example of this would be a bin size of 7 days and a list of daily dates – the same bin would be returned for each week of dates within that bin).

If, on the other hand, the span between successive datetimes in the provided list is larger than the defined bin size, each provided date will be repeatly returned with each bin that exists between the provided datetimes. (An example of this would be a bin size of 7 days and a list of monthly dates – each monthly date would be succesively returned with the 4 (or so) bins touched by that month)

See `make_datetime` for more detail on acceptable formats for date descriptors.

next_date_bin(*date*) → datetime
Returns a `datetime.datetime` object representing the beginning of the date bin following the date bin in which the given date resides.

See `make_datetime` for more detail on acceptable formats for date descriptors.

prior_date_bin(*date*) → datetime
Returns a `datetime.datetime` object representing the beginning of the date bin prior to the date bin in which the given date resides.

See `make_datetime` for more detail on acceptable formats for date descriptors.

today_bin() → datetime
Returns a `datetime.datetime` object representing the beginning of the date bin containing the current date.

netsa.data.times.**dow_day_snapper**(*size : int*[, *dow=0*]) → DateSnapper
Given an integer size in days and an integer day-of-the-week, returns a :class:`DateSnapper` object anchored on the first occurring instance of that DOW after the given epoch, which defaults to the UNIX epoch. Monday is the 0th DOW. DOW values are modulo 7, so the 7th DOW would also represent Monday.

CHAPTER

SEVEN

MISCELLANEOUS FACILITIES

7.1 `netsa.files` — File and Path Manipulation

The routines in `netsa.files` are intended to help with manipulation of files in the filesystem as well as pathnames.

7.1.1 Paths

`netsa.files.`**`relpath`**(*p : str*, *base : str*) → str

Given a target path along with a reference path, return the relative path from the target to the reference.

This is a logical operation that does not consult the physical filesystem.

`os.path.relpath` in Python 2.6 adds something similar to this.

`netsa.files.`**`is_relpath`**(*p : str*, *base : str*) → bool

Given a target path along with a base reference path, return whether or not the base path subsumes the target path.

This is a logical operation that does not consult the physical filesystem.

7.1.2 Process ID Locks

This form of lock is generally used by services that wish to ensure that only one copy of the service is running at a time.

`netsa.files.`**`acquire_pidfile_lock`**(*path : str*) → bool

Attempts to acquire a locking PID file at the requested pathname *path*. If the file does not exist, creates it with the current process ID. If the file does exist but refers to a no-longer-existing process, attempts to replace it. If the file does exist and refers to a running process, does nothing.

Registers the lock to be released (as with `release_pidfile_lock`) when the currently running process exits, if it has not already been released.

Returns `True` if this process holds the lock, or `False` if another running process holds the lock.

`netsa.files.`**`examine_pidfile_lock`**(*path : str*) → (int, bool) or None

Examines the state of a locking PID file at *path* and returns it. If the file does not exist or is not in the proper format, returns `None`. If the file does exist and contains a PID, returns `(pid, state)` where *pid* is the process ID holding the lock, and *state* is `True` if a running process has that process ID, or `False` otherwise.

`netsa.files.`**`release_pidfile_lock`**(*path : str*)

Attempts to release a locking PID file at the requested pathname *path*. If the file does not exist or contains a lock for a different PID, does nothing. Otherwise, unlinks the file.

7.1.3 Temporary Files

netsa.files.**get_temp_file_name**([*file_name : str*]) → str
Return the path to a file named *file_name* in a temporary directory that will be cleaned up when the process exits. If *file_name* is None then a new file name is created that has not been used before.

A temporary file at the named location will be cleaned up as long as the Python interpreter exits normally.

netsa.files.**get_temp_file**([*file_name : str*, *mode='r'*]) → file
Returns an open file object for the file named *file_name* in the temporary working directory, with the given *mode* (as described in open. If *file_name* is None then a new file name is used that has not been used before.

The resulting temporary file will be cleaned up as long as the Python interpreter exits normally.

netsa.files.**get_temp_pipe_name**([*pipe_name : str*]) → str
Returns the path to a named pipe *file_name* that has been created in a temporary directory that will be cleaned up when the process exits. If *file_name* is None then a new file name is created that has not been used before.

7.2 netsa.json — JSON Wrapper Module

The netsa.json module provides a wrapper module for either the Python standard library json module, if it is available, or an included copy of the simplejson module, otherwise. Please see the standard library documentation for details.

7.3 netsa.util.clitest — Utility for testing CLI tools

7.3.1 Overview

The netsa.util.clitest module provides an API for driving automated tests of command-line applications. It doesn't do the work of a test framekwork; for that, use a framework library such as unittest or functest.

Enough of netsa.util.clitest has been implemented to fulfill a minimal set of requirements. Additional features will be added as necessary to support more complex testing.

This module is influenced by http://pythonpaste.org/scripttest/.

A usage example:

```
from clitest import *
env = Environment("./test-output")
# Run the command
result = env.run("ryscatterplot --help")
assert(result.success())
assert(result.stdout() == "whatever the help output is")
assert(result.stderr() == "")
# Clean up whatever detritus the command left
env.cleanup()
```

7.3.2 Exceptions

exception netsa.util.clitest.**TestingException**
Class of exceptions raised by the clitest module.

7.3.3 Classes

class netsa.util.clitest.**Environment**([*work_dir : str*][, *save_work_dir : bool*][, *debug : bool*][, *<env_name>=<env_val>, ...*])

An environment for running commands, including a set of environment variables and a working directory.

The *work_dir* argument is the working directory in which the commands are run. If *work_dir* is `None`, a directory will be made using `tempfile.mkdtemp` with default values.

work_dir must not already exist or `run` will raise a `TestingException`. If *save_work_dir* is `False`, `cleanup` will remove this directory when it is called.

If *debug* is `True`, several debug messages will be emitted on `stderr`.

Any additional keyword arguments are used as environment variables.

get_env(*env_name : str*) → str

Returns the value of *env_name* in the environment. If *env_name* does not exist in the environment, this method returns `None`.

set_env(*env_name : str*, *env_val : str*)

Sets the value of *env_name* in the environment to *env_val*. *env_val* must be a string.

del_env(*env_name : str*)

Removes *env_name* from the environment. If *env_name* doesn't exist, this method has no effect.

get_work_dir() → str

Returns the working directory in which the commands are run.

run(*command : str*[, *<keyword>=<value>, ...*]) → Result

Runs a single command, capturing and returning result information. Keyword arguments are passed to `netsa.util.shell.run_parallel`. See the documentation of that function for an explanation of how such arguments are interpreted.

Returns a `Result` object representing the outcome.

cleanup()

Cleans up resources left behind by the test process.

class netsa.util.clitest.**Result**(*command*, *envars*, *exit_codes*, *stdout*, *stderr*, *debug=False*)

Contains information on a command's exit status and output.

success() → bool

Returns `True` if the exit code of the process was 0. This usually, but not always, indicates that the process ran successfully. Know Your Tool before relying on this function.

exited([*code*]) → bool

Returns `True` if the process exited with the specified exit code. If the exit code is `None` or unsupplied, returns `True` if the process terminated normally (e.g., not on a signal).

exit_status() → int or None

Returns the exit status of the process, if the process exited normally (e.g., was not terminated on a signal). Otherwise, this function returns `None`.

signal() → int or None

Returns the signal on which the process terminated, if the process terminated on a signal. Otherwise, this function returns `None`.

signaled() → bool

Returns `True` if the process terminated on the specified signal. If the signal is `None` or unsupplied, returns `True` if the process terminated on a signal.

format_status() → str
Returns a human-readable representation of how the process exited.

get_status() → int
Returns the raw exit status of the process, as an integer formatted in the style of `os.wait`.

get_stdout() → str
Returns the standard output of the process as a string.

get_stderr() → str
Returns the standard error of the process as a string.

get_info() → str
Returns the information contained in the result as a human-readable string.

7.4 `netsa.util.compat` — Python version compatibility code

The `netsa.util.compat` module provides some additional functionality introduced between Python 2.4 and the latest versions of Python. Obviously new syntax features cannot be supported, but certain utility functions in modules or built-in functions can be added on for the sake of sanity.

The list of provided features is currently small, but is likely to grow over time.

To use the compatibility features, simply import this module:

```
import netsa.util.compat
```

There is no need to import any specific symbols from the module—it will add the symbols directly where needed so that they may be imported as normal. Built-ins will also work wherever used.

The additional functions currently provided by this module are:

- `all`
- `any`
- `heapq.merge`
- `itertools.product`
- `os.path.relpath`

CHAPTER

EIGHT

INTERNAL FACILITIES

The following features are meant primarily for internal NetSA use. If you do make use of them, do so with the understanding that their interfaces are much more likely to change than other APIs in this document.

8.1 `netsa.dist` — Common Installation Procedures

The `netsa.dist` module is intended primarily for NetSA development team, to provide a common set of practices for generating documentation, running tests, and distributing and installing our software. If you are not a member of the NetSA dev team, you're likely to be better served by using the standard `distutils` module or the more powerful setuptools package.

8.1.1 Overview

`netsa.dist` provides a set of extensions to `distutils`, along with an alternative API for specifying the contents of the distribution. The extensions provide for automatic generation of documentation using Sphinx (including PDF, HTML, and man pages), automatic generation of Python-readable version information from configuration metadata, and targets for running automated tests. The alternative API allows for specification of project metadata in a similar style to the metadata of `netsa.script`.

8.1.2 New `setup.py` Commands

When running `setup.py` for a project that uses `netsa.dist`, all of the normal commands (`build`, `install`, `sdist`, etc.) are available, along with the following:

`check` Run all automated tests for the project.

`check_unit` Run automated unit tests for the project.

`check_other` Run all other automated tests for the project.

`gen_version` Generates any "version" files required by the project. See `add_version_file`.

`gen_doc_html` Generates an HTML manual for this project, placing it in `doc/html`. This manual is in the normal style for Python documentation. It is never automatically generated.

`gen_doc_tools_web` Generates an HTML manual for this project and create a tarball out of it, placing the results in `dist/<name>-<version>-doc-web.tar.gz`. This manual is designed for use on the NetSA Tools website, and this command is used only to generate documentation to be deployed at that site.

gen_doc_man Creates generated man pages for the project, placing them under `doc/man`. This is automatically called when generating a source distribution, and the results will be included in the source tarball. When installing, `netsa.dist` will attempt to run this command, but if it fails it will use a pre-generated copy if available.

gen_doc_pdf Generates a PDF manual for this project, placing it in the top level directory. This is automatically called when generating a source distribution, and the resulting manual will be included in the source tarball. This manual is not installed, however, only included with the distribution.

netsa_dist Generates the standard items for a NetSA distribution. Namely, a source code release tarball in `dist/<name>-<version>.tar.gz` and a documentation tarball in `dist/<name>-<version>-doc-web.tar.gz` for deployment to the NetSA Tools website.

netsa_src_license Modifies the source files in place to update their license section. The license in `LICENSE-xxx.txt` is used for a section that begins with `@xxx_HEADER_START@`. Backup files are created, just in case.

8.1.3 Project Layout

The files of the project are expected to be arranged mostly as follows:

Directory	Purpose
bin	Contains Python scripts to be installed as executables.
doc	Contains Sphinx documentation sources, including `conf.py`. See `disable_documentation`, and Documentation Configuration.
src	Contains Python source code and data files to be installed in Python packages. See `add_package`, `add_package_data`, and `add_module_py`.

The following directories are used to contain outputs, and any extra files in them may be automatically destroyed by the `clean` command:

Directory	Purpose
build	A variety of intermediate products are stored here while building the project.
dist	Final products (tarballs) are stored here.
doc/html	HTML documentation generated with `gen_doc_html` will do here.
doc/man	manpage documentation generated with `gen_doc_man` will go here.

8.1.4 Documentation Configuration

To support simpler common configuration for documentation output, a convenience module is create during documentation generation. Under most circumstances, you should be able to use the following `conf.py` without changes:

```
from netsa_sphinx_config import *

add_static_path("static_html")
```

Importing all symbols from `netsa_sphinx_config` sets all of the settings to their normal values for a NetSA project, including producing output appropriate for use on the NetSA Tools website automatically.

If you need to make modifications, just replace or modify the values of standard Sphinx build options.

The following function is provided to allow automatic generation of man pages. Any man page generated from the documentation will be automatically generated and included in source distributions, and automatically installed in the appropriate location.

netsa_sphinx_config.**add_man_page**(*source_file : str*, *man_page : str*, *description : str*[, *section = 1*])

Add a new man page to be generated from the given *source_file* (without extension). The name of the resulting file is *man_page*.*section*. Note that when Sphinx generates man pages, the top-level heading from the input file is ignored, and the title used is "*man_page - description*" instead. This way, you can use the same input file to produce installed man pages and to produce man pages for display in the HTML output.

8.1.5 Project Configuration

The following functions are used to set metadata for the project:

netsa.dist.**set_name**(*project_name : str*)

Sets the name of the project. This name is used as part of the name of produced tarballs and documentation files.

netsa.dist.**set_title**(*project_title : str*)

Sets the title for this project. This should be the human-readable name of the project. It is displayed in most places as the project name.

netsa.dist.**set_description**(*project_description : str*)

Sets the long-form description for this project. This should be a detailed explanation of the project's purpose.

netsa.dist.**set_version**(*project_version : str*)

Sets the version number for this project. The version number is used as part of the filename of distribution files, is included in the documentation, and may be written out as version files (see `add_version_file` and `netsa.find_version`).

netsa.dist.**set_copyright**(*project_copyright : str*)

Sets the copyright date for this project. This is used in documentation generation, and in the project metadata. For example:

```
dist.set_copyright("2008-2011, Carnegie Mellon University")
```

netsa.dist.**set_license**(*project_license : str*)

Sets the license type for this project, which defaults to `'GPL'`. This is used for project distribution metadata.

netsa.dist.**set_maintainer**(*project_maintainer : str*)

Given a name and email address (i.e. `'Harry Q. Bovik <bovik@sample.samp>'`) sets the maintainer name and email address metadata for the project.

netsa.dist.**set_author**(*project_author : str*)

Given a name and email address (i.e. `'Harry Q. Bovik <bovik@sample.samp>'`) sets the author name and email address metadata for the project.

netsa.dist.**set_url**(*project_url : str*)

Sets the home page URL metadata for this project.

netsa.dist.**set_download_url**(*project_download_url : str*)

Sets the download page URL metadata for this project.

Choosing which files should be installed where is accomplished with the following functions:

netsa.dist.**add_package**(*package_name*)

Adds a Python package to be installed, by package name. For example:

```
dist.add_package("netsa.data")
```

The files for this Python package would be found under `src/netsa/dist`. Remember that the package directory (and every directory leading up to it) must include an `__init__.py` file to be accepted as a Python package.

netsa.dist.**add_package_data**(*package_name*, *data_file_glob*)
Adds one or more data files to be installed within a package. Each file or directory that *data_file_glob* expands to is included. The files and directories should be stored under src/<package_name>, just like the Python source files for the package. For a method of installing files in different places, see add_install_data.

netsa.dist.**add_module_py**(*module_name*)
Adds a single module by module name. For example:

```
dist.add_module_py("netsa.util.shell")
```

Thie file for this module would be found at src/netsa/util/shell.py. Remember that the package directory (and every directory leading up to it) must include an __init__.py file, which will also be installed.

netsa.dist.**add_module_ext**(*module_name*, *module_sources*, ***kwargs*)
Adds a single C extension module, given a module name, a list of sources, and optional keyword arguments as accepted by distutils.core.Extension. For example:

```
dist.add_module_ext('foo', ['foo.c', 'bar.c'])
```

netsa.dist.**add_script**(*script_name*)
Adds a single script by script name. For example:

```
dist.add_script("helloworld")
```

The file for this script would be found at bin/helloworld. When installed, if the script has a #! line and contains python, it will automatically be modified to point to the version of Python being used to install this project.

netsa.dist.**add_install_data**(*install_path*, *data_file_name*)
Adds an extra data file that should be installed when the project is installed. The *data_file_name* should be the path to the file from the top level of the project. *install_path* should be the path to the installation directory from the install prefix. For example:

```
dist.add_install_data("share/doc/helloworld", "samples/helloworld.ini")
```

This would install the file found at samples/helloworld.ini as .../share/doc/helloworld/helloworld.ini under the installation prefix.

netsa.dist.**add_extra_files**(*extra_glob*)
Given a glob string, adds files which match that glob to the distribution. This is used to add any extra files (README, etc.) that should be included in a source distribution but are not to be installed. If you do include in this list a file that's already to be installed, it will still be installed, and it will still be included in the distribution.

In order to avoid recording the version number in both the setup.py file and the source code, you can use the following functions to automatically generate a file with the version number in it, and read it back at run time:

netsa.dist.**add_version_file**(*version_file_name : str*[, *version_file_template = "%s\n"*])
Adds a "version file" to the project, with the given path and template. By default the template is "%s\n", which simply includes the version number and a newline. The path should be given relative to the base of the project. The version file will be generated automatically before any other processing is done.

See netsa.find_version for a convenient method for retrieving the version number from this file for your package.

Example:

```
# in setup.py
dist.add_version_file("src/netsa/VERSION")

# in netsa/__init__.py
__version__ = netsa.find_version(__file__)
```

netsa.**find_version**(*source_file : str*[, *num_levels = 3*]) → str
Given the path to a Python source file, read in a version number from a file VERSION in the same directory, or look for a setup.py file in up to num_levels directories above the file and attempt to find the version there.

The following functions allow automated tests to be added and run from `setup.py`:

netsa.dist.**add_unit_test_module**(*script_unit_test_module : str*)
Adds a unit test module to be run, by module name. For example:

```
dist.add_unit_test_module("netsa.data.test")
```

The provided module is expected to be a `unittest` test module, and the tests will be run in a separate process from the process running `setup.py`. Running tests automatically builds the project, and places the build area in the `PYTHONPATH` for the test process.

netsa.dist.**add_other_test_module**(*script_other_test_module : str*)
Adds a module to be run for testing, by module name. For example:

```
dist.add_other_test_module("crunchy.test")
```

The provided module is called in a subprocess like this:

```
python -m crunchy.test ${source_dir}
```

Where `${source_dir}` is the top level source directory of the project. Running tests automatically builds the project, and places the build area in the `PYTHONPATH` for the test process.

Finally, once the project is fully configured, use this function to handle command-line options and actually running the tasks:

netsa.dist.**execute**()
Using the project as so far specified, parse command line options and does what is required to build, install, test, or make a distribution for the project. This should be called as the last thing in `setup.py`.

CHAPTER

NINE

CHANGES

9.1 Version 1.5 - 2016-05-24

- netsa-python no longer supports versions of Python prior to 2.6.
- Removed functions and modules deprecated in netsa-python 1.4.
- Bug fixes and improvements to netsa.script.golem.

9.2 Version 1.4.4 - 2014-01-29

- Fixed error in netsa.script that caused a default for file params to be checked for before command-line arguments were handled.
- Fixed default regex for label params in netsa.script.

9.3 Version 1.4.3 - 2013-02-18

- Added server-side cursor support to netsa.sql's psycopg2 driver, so that results are streamed back instead of fetched all together.
- Fixed an exception caused by using num_prefix on values <= 1e-21.
- Remove netsa_silk support version SiLK versions older than SiLK 3.
- Provide unnamed arguments to scripts via netsa.script.get_extra_args().
- Better support for finding non-standard Sphinx installs.

9.4 Version 1.4.2 - 2012-12-07

- Fix problem where netsa.dist would not install manpages unless the distribution also had install_data.

9.5 Version 1.4.1 - 2012-11-09

- Fail-on-use dummy IPv6Addr added to netsa_silk, so that the symbol may still be imported even when IPv6 support is not provided.

- Minor bug fixes to prevent setup.py from requiring sphinx, and to improve documentation generation on projects with different needs.

9.6 Version 1.4 - 2011-09-30

- NOTE: Version 1.4 of NetSA Python is the last version that will support Python 2.4. Future major versions of NetSA Python will require Python 2.6 or greater.
- Added new netsa_silk module, which provides a bridge between netsa-python and PySiLK. When PySiLK is available, it uses PySiLK to provide a fast C implementation of IP address and related functionality. When PySiLK is not available, a pure Python implementation from netsa-python is used instead.
- Added new netsa.script.golem script automation framework, for building scripts to maintain large time-based data sets (among other things.)
- Added regex_help argument to netsa.script.add_text_param and add_label_param, to support providing a more useful error message when the regex doesn't match the input.
- Added heapq.merge and os.path.relpath to netsa.util.compat.
- Replaced all temporary file code with new functions in the netsa.files module, to avoid duplication of effort. Similar functions in other locations have been deprecated and are now implemented using this version.
- Replaced PID locking code with new functions in the netsa.files module, to avoid duplication of effort.
- Improved netsa.data.nice's results, particularly for time ticks.
- Added a large number of new tests to improve compatibility testing for different versions of Python in the future.
- Moved the netsa.script.get_temp_dir... functions into netsa.files, with slight renaming. The old functions are still available but deprecated.
- Deprecated netsa.files.DirLocker, netsa.files.LocalTmpDir, and netsa.tools.service. See the new functions netsa.files.acquire_pidfile_lock, examine_pidfile_lock, and release_pidfile_lock.
- Deprecated netsa.files.datefiles.
- Added documentation for the netsa.dist module, even though it is primarily for internal use.
- Fixed bug involving SIGPIPE handling in netsa.util.shell.
- Fixed bug that prevented netsa.logging from importing correctly under Python 2.7.

9.7 Version 1.3 - 2011-03-28

- Switched to new common installation mechanism (based on distutils)
- Improved error handling in netsa.util.script
- Added new function netsa.script.get_temp_dir_pipe_name()
- Added timedelta support to netsa.data.times
- Added new netsa.util.compat to activity "compatibility features"

9.8 Version 1.2 - 2011-01-12

- Added support for Oracle in netsa.sql via cx_Oracle
- Added support for multi-paragraph help text in netsa.script

9.9 Version 1.1 - 2010-10-04

- Added experimental DB connection pooling to netsa.sql
- Made netsa.script flow_params –help work when site config file is missing.
- Added netsa.util.shell.run_collect_files
- Fixed a bug with netsa.script.Flow_params.using
- Fixed a bug involving netsa.script missing metadata causing crashes

9.10 Version 1.0 - 2010-09-14

- Added netsa.util.clitest module to support CLI tool testing.
- Added PyGreSQL support to netsa.sql.
- Fixed a bug in netsa.sql db_query parsing code.
- Fixed a bug in netsa.sql database URI parsing code.
- Fixed bugs in netsa.data.nice nice_time_ticks.

9.11 Version 0.9 - 2010-01-19

- First public release.

CHAPTER

TEN

LICENSES

10.1 License for netsa-python

Use of the Network Situational Awareness Python support library and related source code is subject to the terms of the following licenses:

GNU General Public License (GPL) pursuant to Version 2, June 1991
Government Purpose License Rights (GPLR) pursuant to DFARS 252.227.7013,

NO WARRANTY

ANY INFORMATION, MATERIALS, SERVICES, INTELLECTUAL PROPERTY OR OTHER PROPERTY OR RIGHTS GRANTED OR PROVIDED BY CARNEGIE MELLON UNIVERSITY PURSUANT TO THIS LICENSE (HEREINAFTER THE "DELIVERABLES") ARE ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESS OR IMPLIED AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, INFORMATIONAL CONTENT, NONINFRINGEMENT, OR ERROR-FREE OPERATION. CARNEGIE MELLON UNIVERSITY SHALL NOT BE LIABLE FOR INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES, SUCH AS LOSS OF PROFITS OR INABILITY TO USE SAID INTELLECTUAL PROPERTY, UNDER THIS LICENSE, REGARDLESS OF WHETHER SUCH PARTY WAS AWARE OF THE POSSIBILITY OF SUCH DAMAGES. LICENSEE AGREES THAT IT WILL NOT MAKE ANY WARRANTY ON BEHALF OF CARNEGIE MELLON UNIVERSITY, EXPRESS OR IMPLIED, TO ANY PERSON CONCERNING THE APPLICATION OF OR THE RESULTS TO BE OBTAINED WITH THE DELIVERABLES UNDER THIS LICENSE.

Licensee hereby agrees to defend, indemnify, and hold harmless Carnegie Mellon University, its trustees, officers, employees, and agents from all claims or demands made against them (and any related losses, expenses, or attorney's fees) arising out of, or relating to Licensee's and/or its sub licensees' negligent use or willful misuse of or negligent conduct or willful misconduct regarding the Software, facilities, or other rights or assistance granted by Carnegie Mellon University under this License, including, but not limited to, any claims of product liability, personal injury, death, damage to property, or violation of any laws or regulations.

Carnegie Mellon University Software Engineering Institute authored documents are sponsored by the U.S. Department of Defense under Contract FA8702-15-D-0002. Carnegie Mellon University retains copyrights in all material produced under this contract. The U.S. Government retains a non-exclusive, royalty-free license to publish or reproduce these documents, or allow others to do so, for U.S. Government purposes only pursuant to the copyright license under the contract clause at 252.227.7013.

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U