NetSA Python Documentation

Release 0.9

Carnegie Mellon University

October 04, 2010

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_data 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 `ParamError`

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 UserError
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 ScriptError
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.

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.

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.

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.

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>
```

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`.

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.

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}")
```

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)
```

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.

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`.

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}")
```

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.

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.

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`.

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`.

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`.

get_param(*name : str*)
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`.

1.5 Verbose Output

get_verbosity()
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.

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.

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`.

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`.

get_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 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()
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()
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()
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()
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()
Returns the rwfilter pull --class argument as a str.

get_end_date()
Returns the `rwfilter` pull `--end-date` argument as a `datetime.datetime` object.

get_filenames()
Returns any files given on the command-line for an `rwfilter` pull as a `str`.

get_flowtypes()
Returns the `rwfilter` pull `--flowtypes` argument as a `str`.

get_input_pipe()
Returns the `rwfilter` pull `--input-pipe` argument as a `str`.

get_sensors()
Returns the `rwfilter` pull `--sensors` argument as a `list` of `str`.

get_start_date()
Returns the `rwfilter` pull `--start-date` argument as a `datetime.datetime` object.

get_type()
Returns the `rwfilter` pull `--type` argument as a `str`.

get_xargs()
Returns the `rwfilter` pull `--xargs` argument as a `str`.

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

is_pull()
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]*)
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."

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.) Unlike most parameters, output file parameters never have default values, and 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.

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.

get_output_file_name(*name : 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`.

get_output_file(*name : str*)
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.

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.

get_output_dir_file_name(*dir_name : str, file_name : str, [description : str, mime_type : 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`.

get_output_dir_file(*dir_name : str, file_name : str, [description : str, mime_type : str]*)
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 Temporary Files

get_temp_dir_file_name(*[file_name : 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.

get_temp_dir_file(*[file_name : str, append=False]*)
Returns an open `file` object for the file named *file_name* in the script's temporary working directory. If *append* is `True`, the file is opened for append. Otherwise, the file is opened for write. If *file_name* is `None` then a new file name is used that has not been used before.

1.9 Script Execution

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.SQL — SQL DATABASE ACCESS

2.1 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.)

2.2 Exceptions

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

exception sql_no_driver_exception
This exception is raised when no driver is installed that can handle a URL opened via db_connect.

exception sql_invalid_uri_exception
This exception is raised when the URI passed to db_connect cannot be parsed.

2.3 Connecting

db_connect (*uri, [user : str, password : str]*)
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.

2.4 Connections and Result Sets

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

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

clone ()
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>, ...]*)
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()
Returns which variant tags are associated with this connection.

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

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

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

__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.

2.5 Compiled Queries

class 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>, ...]*)
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*)
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*)
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*)
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*)
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*)
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*)
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.

2.6 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 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*)
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*)
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.

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.

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

2.7 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*.

db_create_pool(*uri, [user : str, password : str], ...*)
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 db_pool()
A pool of database connections for a single specific connection specification and pool configuration. See db_create_pool.

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

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

class db_driver()

create_pool(*uri, user : str or None, password : str or None, ...*)
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.

2.8 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

THREE

NETSA.UTIL.SHELL — ROBUST SHELL PIPELINES

3.1 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"])
```

3.2 Exceptions

exception `PipelineException`

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.

3.3 Building Commands and Pipelines

`command`(*<command spec>, [stderr : str or file, stderr_append=False, ignore_exit_status=False, ignore_exit_statuses : int seq]*)

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:

```
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:

```
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:

```
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:

```
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:

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

Example: Define a command using individual string arguments:

```
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.

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

```
new_pipeline = pipeline(["ls /etc", "tac"])
```

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:

```
new_pipeline = pipeline("ls /etc", "tac")
```

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:

```
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:

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

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

```
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:

```
["ls /tmp/die", "xargs rm"]  =>  pipeline(["ls /tmp/die", "xargs rm"])
["</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:

```
pipeline("ls -lR", ">%(output_file)s")
pipeline("ls -lR", stdout="%(output_file)s")
```

3.4 Running Pipelines

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:

```
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.)

```
run_parallel("mkfifo test.fifo")
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.
```

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

```
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]]
```

`run_collect` (*<command spec>, ..., [vars : dict, ...]*)

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*:

```
# Reverse sort the output of ls -1
(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:

```
# The same with a named directory
(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*:

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

run_collect_files(<*command spec*>, ..., *[vars : dict, ...]*)

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:

```
(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

FOUR

DATA MANIPULATION

4.1 `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 January 2010.

`get_area_numeric`(*code : int or str*)
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.

`get_area_name`(*code : int or 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.

`get_area_tlds`(*code : int or str*)
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.

`get_country_numeric`(*code : int or str*)
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.

`get_country_name`(*code : int or str*)
Given a country code as a string or integer, returns the name for the country.

Raises `KeyError` if the country code is unrecognized.

get_country_alpha2(*code : int or 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.

get_country_alpha3(*code : int or 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.

get_country_tlds(*code : int or str*)
Given a country code as a string or integer, returns a list of zero or moreDNS top-level domains for that country.

Raises `KeyError` if the country code is unrecognized.

iter_countries()
Returns an iterator which yields all known ISO 3166-1 numeric country codes as integers, including user-assigned code elements in use.

get_region_numeric(*code : int or str*)
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.

get_region_name(*code : int or str*)
Given a region code as a string or integer, returns the name for the region.

Raises `KeyError` if the region code is unrecognized.

get_region_tlds(*code : int or str*)
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.

iter_regions()
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.

iter_region_subregions(*code : int or str*)
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.

iter_region_countries(*code : int or str*)
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.

4.2 `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.

4.2.1 Numbers

num_fixed(*value : num, [units : str, dec_fig=2, thousands_sep : 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'
```

num_exponent (*value : num, [units : str, sig_fig=3]*)

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'
```

num_prefix (*value : num, [units : str, sig_fig=3, use_binary=False, thousands_sep : 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(12345678901234567890123456, 's')
```

```
'1.23e+24s'
>>> num_prefix(0.001, 's')
'1.00ms'
>>> num_prefix(0.001, 's', use_binary=True)
'1.00ms'
```

4.2.2 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`.

`datetime_silk`(*value : datetime, [precision=DATETIME_SECOND]*)

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'
```

`datetime_silk_hour`(*value : datetime*)

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'
```

`datetime_silk_day`(*v : datetime*)

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'
```

datetime_iso(*value : datetime, [precision=DATETIME_SECOND]*)

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'
```

datetime_iso_day(*value : datetime*)

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'
```

datetime_iso_basic(*value : datetime, [precision=DATETIME_SECOND]*)

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'
```

4.3 `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.

`nice_ticks`(*lo : num, hi : num, [ticks=5, inside=False]*)

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`.

`nice_ticks_seq`(*lo : num, hi : num, [ticks=5, inside=False]*)

A convenience wrapper of `nice_ticks` to return the nice range as a sequence.

`nice_time_ticks`(*lo : datetime, hi : datetime, [ticks=5, inside=False, as_datetime=True]*)

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.

nice_time_ticks_seq(*lo : datetime, hi : datetime, [ticks=5, inside=False, as_datetime=True]*)
A convenience wrapper of nice_time_ticks to return the nice range as a sequence.

4.4 netsa.data.times — Time and Date Manipulation

Backwards-compatibility binding for netsa.data.times. New code should be sure to use netsa.data.times directly instead.

make_datetime(*v : num or str or datetime or mxDateTime, [utc_only=True]*)
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.

bin_datetime(*dt : timedelta, t : datetime, [z=UNIX_EPOCH : 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.

4.4.1 Date Snappers

class DateSnapper(*size : timedelta, [epoch=UNIX_EPOCH : datetime]*)
Class for date bin manipulations

date_aligned(*date*)
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*)
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*)
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*)
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*)
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*)
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*)
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*)
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()
Returns a `datetime.datetime` object representing the beginning of the date bin containing the current date.

dow_day_snapper(*size : int, [dow=0]*)
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 UNIX epoch. Monday is the 0th DOW. DOW values are modulo 7, so the 7th DOW would also represent Monday.

CHAPTER

FIVE

MISCELLANEOUS FACILITIES

5.1 `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.

5.1.1 Paths

`relpath`(*p : str, base : 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.

`is_relpath`(*p : str, base : str*)
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.

5.1.2 Directory-based Locking

exception `DirLockBlock`
Raised when an attempt to lock a directory is blocked for too long a time by another process holding the lock.

class `DirLocker`(*name : str, [dir : str, seize=False, debug=False]*)
Provides cheap cross-process locking via `mkdir`/`rmdir`. *name* is the token identifying the application group. *dir* optionally specifies the directory in which to establish the lock (defaults to `'/var/tmp'` or some sensible temp dir name). If *seize* is `True` and the requested lock appears to have been orphaned, a new lock is established and the old lock debris is removed.

This is not an infallible locking solution. This is advisory locking. It is possible to have an orphaned lock or a ghosted lock.

For simple scenarios such as avoiding long-running cron jobs from trampling over one another, it's probably sufficient.

See also `netsa.tools.service.check_pidfile`.

5.1.3 Garbage Collected Temporary Directories

exception `LocalTmpDirError`
Raised when an unrecoverable error occurs within `LocalTmpDir`.

class `LocalTmpDir` (*[dir : str, prefix='tmp', create=True, verbose=False, autodelete=True]*)
Provides ephemeral temporary directories, similar to `tempfile.NamedTemporaryFile`. The resulting directory and all of its contents will be unlinked when the object goes out of scope.

The parameter *prefix* is passed as the *prefix* parameter to `tempfile.NamedTemporaryFile` when temporary files are created within the temporary directory.

The parameter *create* controls whether the temporary directory is actually created. If you want to create the directory manually, you can use the method `assert_dir`.

The parameter *verbose* controls whether status messages (such as creation/deletion of files and dirs) are printed to `stderr`.

The parameter *autodelete* controls whether the temporary directory, and all its contents, are deleted once the `LocalTmpDir` object goes out of scope. (mostly useful for debugging)

`assert_dir`()
Checks to see if this temp dir exists and creates it if not. Normally this happens during object creation.

`prefix`()
Returns the value of 'prefix' that is passed to `tempfile.NamedTemporaryFile`.

`tmp_file`()
Returns a `tempfile.NamedTemporaryFile` object within within this temp dir.

`tmp_filename`()
Returns a new temporary filename within this temp dir.

`tmp_pipe`()
Returns the filename of a new named pipe within within this temp dir.

5.2 `netsa.files.datefiles` — Date-based filenames

5.2.1 Exceptions

exception `DateFileParseError`
Raised if a function is unable to parse a date within the provided filename.

5.2.2 Filename Manipulation

`date_from_file` (*file : str*)
Attempt to extract dates from a filename. The filename can be a full pathname or relative path. Dates are presumed to exist somewhere in the pathname. See `split_on_date` for more detail on how dates are parsed from filenames.

`split_on_date` (*file : str*)
Given a string (presumably the pathname to a file) with a date in it, return the directory of the file and the date/non-date components of the file name as an array. This routine is pretty liberal about what constites a valid date format since it expects, by contract, a filename with a date string embedded within it.

For example, the input `"foo/bar-20090120:12:17:21.txt"` parses to:

```
('foo', ['bar-', 2009, None, 1, None, 20, ':", 12, ':', 17, ':', 21, '.txt'])
```

The following are all valid dates:

```
2008
200811
20081105
2008/11
2008/11/05
2008-11-05
2008.11.05
2008-11-05:07
2008-11-05:07:11
2008-11-05:07:11:00
```

Separators between year/month/day are 'non digits'. This implies that directories within the path string can contribute to the date along with information in the filename itself. The following is valid:

```
'/path/to/2008/11/05.txt'
```

The extraction is non-greedy: only the 'last' part that looks like a date is extracted. For example:

```
'/path/to/2008/11/2008-11-05.txt'
```

extracts only '2008-11-05' from the end of the string.

Separators between hour/minute/sec must be ':'

date_file_template(*file : str, [wildcard='x']*)
Given a pathname *file*, returns the string with `'x'` in place of date components. The replacement character can be overridden with the *wildcard* argument.

This is useful for determining what dated naming series are present in a shared directory.

sibling_date_file(*file : str, date : datetime*)
Given a filename *file*, along with a date, returns the analagous filename corresponding to that date.

5.2.3 Directory Walking

datefile_walker(*dir : str, [suffix : str, silent=False, snapper : DateSnapper, descend=True, reverse=False]*)
Returns an iterator based on the dated files that exist in directory *dir*. Each value returned by the iterator is a tuple of `(date, [file1, file2, ...])`, where each file matches the given date. See `split_on_date` for more detail on how dates are parsed from filenames.

If *descend* is `True`, the entire directory tree will be traversed. Otherwise, only the top-level directory is examined.

If *reverse* is `True`, the iterator returns entries for each date in descending order. Otherwise, entries are returns in ascending order.

If a `netsa.data.times.DateSnapper` *snapper* is provided, it will be used to enforce the alignment of dates, throwing a `ValueError` if a misaligned date is encountered.

latest_datefile(*dir : str, [suffix : str, silent=False, snapper : DateSnapper, descend=True]*)
Traverses the given directory and returns a single tuple `(date, [file1, file2, ...])` where date is the latest date present and each file in the list contains that date. See `split_on_date` for more detail on how dates are parsed from filenames.

If *descend* is `True`, the entire directory tree is traversed recursively. Otherwise, only the top-level directory is examined.

If a `netsa.data.times.DateSnapper` *snapper* is provided, it will be used to enforce the alignment of dates, throwing a `ValueError` if a misaligned date is encountered.

date_snap_walker (*dir : str, snapper : DateSnapper, [suffix : str, sparse=True]*)

Returns an iterator based on traversing the given directory. Each value returned by the iterator is a tuple `(date_bin, ((date, file), (date2, file2), ...))` where each filename contains a date which falls within the given date bin (as defined by *snapper*).

The beginning and ending dates for this sequence are determined by what files are present on the system. If *spare* is `True`, then only date bins which are actually occupied by files in the directory are emitted. Otherwise, a tuple is generated for each date between the smallest and largest dates present in the directory. See `netsa.data.times.DateSnapper` for more information on date bins.

If *suffix* is provided, all files not ending with the provided extension are ignored.

tandem_datefile_walker (*sources : str seq, [suffix : str, silent=True, snapper : DateSnapper, reverse=False]*)

Returns an iterator based on traversing multiple directories given by *sources*. Each value returned by the iterator is a tuple `(date, (dir, file), (dir2, file2), ...)`, where the given directory contains the given file, which contains this date in its name. See `split_on_date` for more detail on how dates are parsed from filenames.

For example, given: `('/dir/one', /dir/two')`, a returned tuple might look like:

```
(date, ('/dir/one', file_from_dir_one_containing_date_in_its_name),
       ('/dir/two', file_from_dir_two_containing_date_in_its_name),
       ('/dir/two', another_file_from_dir_two_with_date_in_its_name))
```

If *suffix* is provided, all files not ending with the provided extension are ignored.

If *reverse* is `True`, tuples are generated with dates in descending order. Otherwise, dates are generated in ascending order.

If a `netsa.data.times.DateSnapper` *snapper* is provided, it will be used to enforce the alignment of dates, throwing a `ValueError` if a misaligned date is encountered.

5.3 `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.

5.4 `netsa.tools.service` — Tools for building services

check_pidfile (*path : str, [unlink=True]*)

Attempts to create a locking PID file at the requested pathname *path*. If the file does not exist, creates it with the current process ID, and sets up an `atexit` process to remove it. If the file does exist but refers to a no-longer-existing process, replaces it and does the above. If the file does exist and refers to a running process, does nothing.

Returns `True` if the PID file was created or replaced (which means we should continue processing), or `False` if the PID file was left in place (which means someone else is processing and we should exit.)

5.5 netsa.util.clitest — Utility for testing CLI tools

5.5.1 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()
```

5.5.2 Exceptions

exception TestingException
Class of exceptions raised by the clitest module.

5.5.3 Classes

class 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*)
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 ()
Returns the working directory in which the commands are run.

run(*command : str, [<keyword>=<value>, ...]*)
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 Result(*command, envars, exit_codes, stdout, stderr, debug=False*)
Contains information on a command's exit status and output.

success()
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]*)
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()
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()
Returns the signal on which the process terminated, if the process terminated on a signal. Otherwise, this function returns `None`.

signaled()
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()
Returns a human-readable representation of how the process exited.

get_status()
Returns the raw exit status of the process, as an integer formatted in the style of `os.wait`.

get_stdout()
Returns the standard output of the process as a string.

get_stderr()
Returns the standard error of the process as a string.

get_info()
Returns the information contained in the result as a human-readable string.

CHAPTER

SIX

CHANGES

6.1 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.

6.2 Version 0.9 - 2010-01-19

- First public release.

CHAPTER

SEVEN

LICENSES

7.1 License for netsa-python

Copyright 2008-2010 by Carnegie Mellon University

Use of the Network Situational Awareness Python support library and related source code is subject to the terms of the following licenses: GNU Public License (GPL) Rights 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 F19628-00-C-0003. 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.

7.2 License for simplejson

Copyright (c) 2006 Bob Ippolito

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.