NetWorkSpaces for Python
User Guide
Version 1.6

Scientific Computing Associates, Inc.
http://www.lindaspaces.com

June 22, 2007

Contents

Chapter 1

Introduction

NetWorkSpaces (NWS) provides a framework to coordinate programs written in scripting languages; NWS currently supports the languages Python, Matlab, and R. This User Guide is for the Python language, and it assumes that the reader is familiar with Python.

1.1 Data Sharing

A Python program uses variables to communicate data from one part of the program to another. For example, `x = 123` assigns the value 123 to the variable named `x`. Later portions of the program can reference `x` to use the value 123. This mechanism is generally known as *binding*. In this case, the binding associates the value 123 with the name `x`. The collection of name-value bindings in use by a program is often referred to as its *workspace*.

Two or more Python programs use NWS to communicate data by means of name-value bindings stored in a network-based workspace (a NetWorkSpace, which in Python is an instance of a NetWorkSpace object). One program creates a binding, while another program reads the value the binding associated with the name. This is clearly quite similar to a traditional workspace; however, a NetWorkSpace differs from a traditional workspace in two important ways.

First, in a setting in which two or more Python programs are interacting, it would not be unusual for one to attempt to "read" the value of a name before that name

has been bound to a value. Rather than receiving an "unbound variable" error, the reading program (by default) simply waits (or "blocks") until the binding occurs. Second, a common usage paradigm involves processing a sequence of values for a given name. One Python program carries out a computation based on the first value, while another might carry out a computation on the second, and so on. To facilitate this paradigm, more than one value may be bound to a name in a workspace and values may be "removed" (fetch) as opposed to read (find). By default, values bound to a name are consumed in first-in-first-out (FIFO) order, but other modes are supported: last-in-first-out (LIFO), multiset (no ordering implied) and single (only the last value bound is retained). Since all its values could be removed, a name can, in fact, have no values associated with it.

A NetWorkSpace provides five basic operations: **store**, **fetch**, **fetchTry**, **find**, and **findTry**:

store introduces a new binding for a specific name in a given workspace.

fetch fetches (removes) a value associated with a name, blocking if necessary until a value is available.

fetchTry fetches (removes) a value associated with a name without blocking.

find reads a value without removing it, blocking if necessary.

findTry reads a value without removing it and without blocking.

Note that **fetchTry** and **findTry** return `None` or a user-supplied default if no value is available.

There are several additional NetWorkSpace operations:

currentWs returns the name of the specified workspace.

declare declares a variable name with a specific mode.

deleteVar deletes a name from a workspace.

listVars provides a list of variables (bindings) in a workspace.

In addition to a NetWorkSpace, a Python client of NWS also uses an NWSServer object. This object is created automatically when a new NetWorkSpace object is created, so you don't need to interact directly with it. However, `server` is an attribute of NetWorkSpace objects, providing access to the NwsServer object:

```
>>> ws = NetWorkSpace('foo')
```

```
>>> ws.server
```

A NWSServer object supports the following actions:

openWs connects to a workspace or creates one if the specified workspace does not exist.

useWs uses a NetWorkSpace without claiming ownership.

deleteWs explicitly deletes a workspace.

listWss provides a list of workspaces in the server.

close closes the connection to a workspace. Depending on the ownership, closing a connection to a workspace can result in removing the workspace.

1.2 Parallel Computing

The operations above enable coordination of different programs using NWS. There is also a mechanism, built on top of NWS, called Sleigh (inspired by R's SNOW package) to enable parallel function evaluation. Sleigh is especially useful for running *embarrassingly parallel* programs on multiple networked computers. Once a sleigh is created by specifying the nodes that are participating in computations, you can use:

eachElem is used to execute a specified function multiple times in parallel with a varying set of arguments.

eachWorker is used to execute a function exactly once on every worker in the sleigh with a fixed set of arguments.

Various data structures (workspaces, name-value bindings, etc.) in a NWS server can be monitored and even modified using a web interface. In distributed programming, even among cooperating programs, the state of shared data can be hard to understand. The web interface presents a simple way of checking current state remotely. This tool can also be used for learning and debugging purposes, and for monitoring "normal" Python programs as well.

Chapter 2

Getting Started

This chapter provides procedures for setting up NetWorkSpaces and Sleigh for Python. See the INSTALL and README files in the NetWorkSpaces distribution for more information.

2.1 Prerequisites

NetWorkSpaces runs on Linux, Max OS X, Windows, and most Unix systems. The Python NWS API has no prerequisites, other than Python 2.2 or later. However, you must be running a NetWorkSpaces server on some network accessible machine (which includes the local machine, of course). The server is a Python/Twisted application, and therefore requires Python with Twisted installed. NetWorkSpaces uses Twisted Core and Twisted Web, and Twisted in turns requires Zope Interfaces, which comes bundled with the Twisted installation.

To summarize:

- Python 2.2 or later `http://www.python.org`
- NetWorkSpaces server `http://nws-py.sourceforge.net`
- Twisted Core 2.1 or later `http://twistedmatrix.com`
- Twisted Web 0.5 or later
- Zope Interfaces (as distributed with Twisted)

Note on Windows, NetWorkSpaces has an "all-in-one" installer that will install everything that you need to run the server, Python NWS API, and R NWS API, so if you choose to use that, you don't have to be aware of these software requirements.

2.2 NetWorkSpaces Server

2.2.1 Starting the Server

There are several ways to start a NetWorkSpaces server. If you install NetWorkSpaces on Windows with the "all-in-one" installer, than the server is already running as the "NwsService". You can check on it's status, start, and stop it using the standard Windows tools.

- twistd command (as daemon):

  ```
  % twistd -y /etc/nws.tac
  ```

  Note: nws.tac is installed in different directories, depending on the platform and the type of installation (root versus non-root). For root installation, nws.tac is located in `/etc` on UNIX, and in the PYTHON24 or PYTHON25 directory on Windows.

- nws script (UNIX only):
  The nws script is intended to be an init script, but it can be used manually, as follows:

  ```
  % nws start
  ```

- NwsService (Windows only):
  If you didn't install NetWorkSpaces using the "all-in-one" installer, the procedure installing and running the service is:

  1. Open up a command prompt
  2. Install NwsService by executing NwsService.py, which is located in Python's scripts directory

     ```
     C:\python25\scripts> python NwsService.py install
     ```

  3. Start NwsService

     ```
     C:\> sc start nwsservice
     ```

2.2.2 Stopping the Server

There are several ways to stop the NetWorkSpaces server:

- If started via twistd:
  Go the directory where twistd was executed and type:

  ```
  % kill `cat twistd.pid`
  ```

- If started via nws script:

  ```
  % nws stop
  ```

- If started via NwsService:

  ```
  C:\> sc stop nwsservice
  ```

2.3 NetWorkSpaces Client

2.3.1 Installing the Client

NetWorkSpaces for Python is distributed as a Python source distribution for Posix systems, using the standard Python distribution utilities. The full installation instructions are in the INSTALL file that is included in the source distribution, but here's a quick summary of the "System Installation" for Linux and Mac OS X:

```
% tar xzvf nwsclient_1.x.y.tar.gz
% cd nwsclient_1.x.y
% python setup.py build
% sudo python setup.py install
```

Note the use of the `sudo` command to install this as root. If you don't want to (or can't) do that, you can use some options to install into a different location. One option is a "home" install: command:

```
% python setup.py install --home $HOME
```

which installs some scripts in $HOME/bin, python modules in $HOME/lib/python, and the Twisted TAC file in $HOME/nws.tac. You'll need to set the PYTHONPATH environment variable to $HOME/lib/python in this case.

To get help on the different installation options, use the command:

```
% python setup.py install --help
```

On Windows, the server and client are distributed as binary installations, distributed as EXE files. After downloading them, you simply execute them and follow the instructions on the screen. You can also use the (previously mentioned) "all-in-one" installer.

2.3.2 Start NetWorkSpaces Client

Once you've got a NetWorkSpace server up and running, you're ready to use NetWorkSpaces.

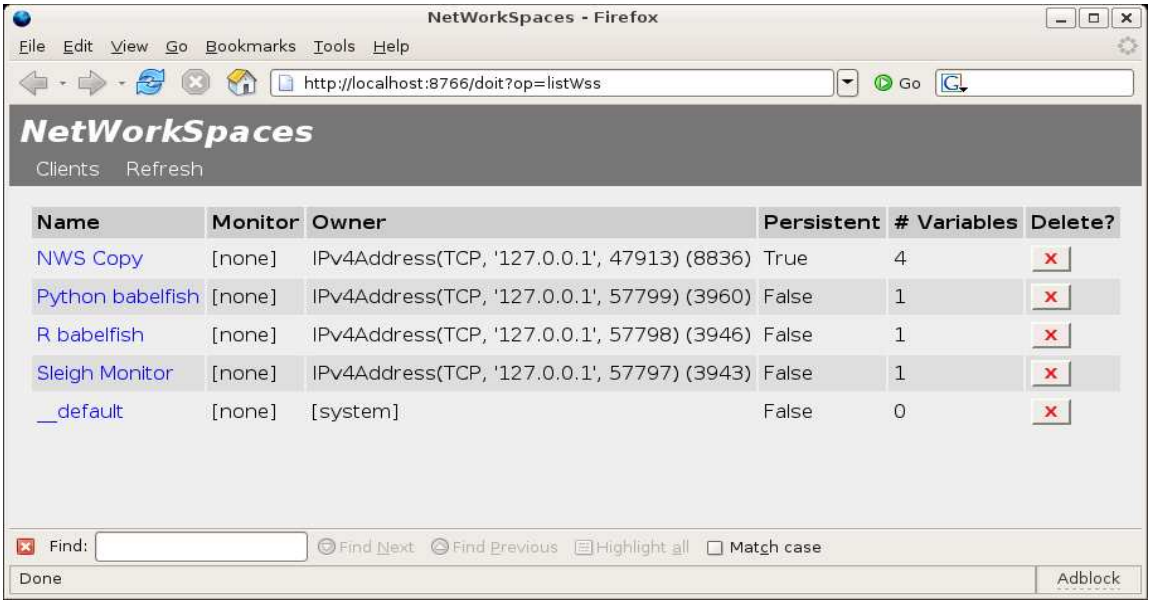
1. Start up a Python session.
2. Type the following:

```
>>> from nws.client import NetWorkSpace
>>> ws = NetWorkSpace('Python space')
>>> ws.store('x', 1)
```

This step creates a workspace named 'Python space' and stores a variable `x` with value 1 to the workspace.

If you've encountered an error importing the NetWorkSpace class, it's likely that you didn't set up PYTHONPATH correctly (assuming you used the "home" install option).

You can also view what's in the workspace using a web interface. To do this, you point your browser to http://*nwsserver*:8766, where *nwsserver* is the machine where the NetWorkSpaces server resides. It should look something like this:



In order to examine the values that you've created from Python in a workspace using the server's web interface, you'll usually need to have the Python babelfish running. The babelfish translates values into a human readable format so they can be displayed in a web browser. If a value is a string, then the web interface simply displays the contents of the string, without any help from the babelfish, but if the value is any other type of Python object, it needs help the help of the Python babelfish.

The Python babelfish is named "pybabelfish", and is installed in the appropriate directory with the other Python and shell scripts, depending on your Python system and how you installed NetWorkSpaces for Python. Assuming that the directory is in your PATH, you could execute the babelfish with the command:

```
% pybabelfish > babelfish.log 2>&1 &
```

or, if you're using a csh compatible shell:

```
% pybabelfish >& babelfish.log &
```

On Windows, the "all-in-one" installer automatically installs and runs the babelfish service, named "PyBabelfishService".

If the NetWorkSpaces server isn't running on the local machine, the pybabelfish `-h` option can be used to specify the correct host name. If the server isn't using the default port (8765), the `-p` option can be used to specify the correct port.

For Posix systems, there is also a version of the babelfish command called "pybabelfishd", which is the daemon version. It must be executed as root, by default. It automatically puts itself into the background, so it is simply run as:

```
% sudo pybabelfishd
```

The log file is created as `/var/log/babelfish.log`. After it starts, pybabelfishd sets it's uid and gid to the "daemon" user. The `-h` and `-p` options (as described above) are also supported.

For more examples on using NetWorkSpaces, see the Tutorials chapter.

2.4 Starting Sleigh

Sleigh is a Python module, built on top of NetWorkSpaces, that makes it very easy to write simple parallel programs. Sleigh has a concept of one master and multiple workers. The master sends tasks to the workers who may or may not be on the same machine as the master. To enable the master to communicate with workers, Sleigh supports several mechanisms to launch workers which execute tasks.

2.4.1 Local Launch Mechanism

The local launch mechanism is the default launch option, and is used for starting workers on the local machine. By default, local launch starts three workers, but you can specify the number of workers to launch with the `workerCount` option to the sleigh constructor. Local launch is useful for developing NetWorkSpace programs, but it's also the best choice when running on a multicore/multiprocessor computer.

Here's how to create a Sleigh object:

```
>>> from nws.sleigh import Sleigh
>>> s = Sleigh()
```

This is equivalent to:

```
>>> s = Sleigh(launch='local')
```

2.4.2 SSH Launch Mechanism

To start up workers on machines other than your local computer, you need to use a remote execution mechanism. SSH is the most common method on Unix-like systems. To use SSH with NetWorkSpaces, the nws.sleigh module supplies a function called `sshcmd`. You simply pass `sshcmd` to the Sleigh constructor using the `launch` option, and specify the nodes to run on with the `nodeList` option.

For example, here is how you starts workers on the machines 'node1' and 'node2':

```
>>> from nws.sleigh import Sleigh, sshcmd
>>> s = Sleigh(launch=sshcmd, nodeList=['node1', 'node'2'])
```

You should first configure ssh so that it won't prompt the user for a password. This is normally done using the *ssh-agent* program. See the Appendix B for more information on this.

We don't really recommend using `sshcmd` on Windows, as different ssh implementations use various tricks that cause quoting problems that are very difficult to diagnose. Nevertheless, we have had some success using Cygwin's ssh server and copSSH from ITef!x, `http://www.itefix.no/phpws/`. If you really want to give it a try, at least avoid using any directories with spaces in them.

2.4.3 RSH Launch Mechanism

RSH can also be used to launch the workers, but it is very insecure, and SSH should always be preferred.

To start sleigh workers using RSH, the procedure is almost the same as with SSH:

```
>>> from nws.sleigh import rshcmd
>>> s = Sleigh(nodeList=['node1', 'node2'], launch=rshcmd)
```

2.4.4 Web Launch Mechanism

Web launch mechansim allows user to start workers on different machines in an ad-hoc way, without remote login mechanisms, such as SSH and RSH.

To use the web launch option, follow the steps below.

1. Create an instance of Sleigh:

   ```
   >>> s = Sleigh(launch='web')
   ```

   The Sleigh constructor does not return until it gets a signal that all workers have started and are ready to accept jobs.

2. Log in to a remote machine.

3. Start a Python session.

4. Open a web browser and point to http://*nwsserver*:8766

5. Click on the newly created Sleigh workspace, and read the value from variable 'runMe'. It usually has value similar to:
   ```
   import nws.sleigh;
   nws.sleigh.webLaunch('sleigh_ride_004_tmp', 'n1.xyz.com', 8765)
   ```

6. Copy the 'runMe' value to the Python session.

7. Repeat steps 2-6 for each worker that needs to be started.

8. Once all workers have started, delete the 'DeleteMeWhenAllWorkers Started' variable from the Sleigh workspace. This signals Sleigh master that the workers have started and are ready to accept work.

Chapter 3

Tutorials

3.1 Data Sharing Tutorial

NetWorkSpaces (NWS) is a Python package that makes it very easy for different Python programs running on (potentially) different machines to communicate and coordinate with one another.

To get started with Python NWS, you'll first have to install the NWS server and the Python NWS client, as described in the Getting Started chapter.

Next, an NWS server must be started. This can be done from a shell using the `twistd` command, as follows:

```
% twistd -y /etc/nws.tac
```

This starts the server as a daemon process, and creates a log and pid file in the current working directory.

Now start an interactive Python session, and import the NetWorkSpace class:

```
% python
>>> from nws.client import NetWorkSpace
```

Next, create a workspace called 'bayport':

```
>>> ws = NetWorkSpace('bayport')
```

This is using the NWS server on the local machine. Additional arguments can be used to specify the hostname and port used by the NWS server if necessary.

Once we have a workspace, we can write data into a variable using the *store* method:

```
>>> ws.store('joe', 17)
```

The variable `joe` now has a value of 17 in our workspace. To read that variable we use the *find* method:

```
>>> age = ws.find('joe')
```

which assigns 17 to `age`.

Note that the *find* method will block until the variable `joe` has a value. That is important when we're trying to read that variable from a different machine. If it didn't block, you might have to repeatedly try to read the variable until it succeeded. Of course, there are times when you don't want to block, but just want to see if some variable has a value. That is done with the *findTry* method.

Let's try reading a variable that doesn't exist using *findTry*:

```
>>> age = ws.findTry('chet')
```

That assigns `None` to `age`, since we haven't stored any value to that variable. If you'd rather have *findTry* return some other value when the variable doesn't exist (or has no value), you can use the command:

```
>>> age = ws.findTry('chet', 0)
```

which assigns 0 to `age`.

So far, we've been using variables in workspaces in much the same way that global variables are used within a single program. This is certainly useful, but NWS workspaces can also be used to send a sequence of messages from one program to another. If the *store* method is executed multiple times, the new values don't overwrite the previous values, they are all saved. Now the *find* method will only be able to read the first value that was written, but this is where another method called *fetch* is useful. The *fetch* method works the same as *find*, but in addition, it removes the value.

Let's try to write multiple values to a variable:

```
>>> n = [16, 19, 25, 22]
>>> for i in n:
...     ws.store('biff', i)
...
```

To read the values, we just call *fetch* repeatedly:

```
>>> n = []
> for i in range(4):
...   n.append(ws.fetch('biff'))
...
```

If we didn't know how many values were stored in a variable, we could have done the following:

```
>>> n = []
>>> while 1:
...     t = ws.fetchTry('biff')
...     if not t: break
...     n.append(t)
...
```

This uses *fetchTry*, which works like *fetch*, except that it is non-blocking, just like *findTry*.

These are the basic operations provided by NWS. It's a good idea to play around with these operations using two Python sessions. That way, you can really transfer data between two different programs. Also, you can see how the blocking operations work. We were pretty careful never to block in any of the examples above because we were only using one Python session.

To use two Python sessions, just execute Python in another window, import NetWorkSpace, and then open the 'bayport' workspace. This can be done with the same commands that we used previously, but this time, the command:

```
>>> ws = NetWorkSpace('bayport')
```

won't create the bayport workspace, since it already exists.

Now you can execute an operation such as:

```
>>> x = ws.fetch('frank')
```

in one session, watch it block for a minute, and then execute *store* in the other session:

```
>>> ws.store('frank', 18)
```

and see that the *fetch* in the first session completes.

While you're experimenting with these operations, it can be very helpful to use the NetWorkSpace server's web interface to see what's going on in your workspaces. Just point a web browser to: `http://localhost:8766`.

If you're using a browser on a different machine from the NWS server you'll have to use the appropriate hostname, rather than 'localhost'.

3.2 Sleigh Tutorial

Sleigh is a Python package, built on top of NWS, that makes it easy to write parallel programs. It provides two basic functions for executing tasks in parallel: *eachElem* and *eachWorker*.

eachElem is used to execute a specified function multiple times in parallel with a varying set of arguments.

eachWorker is used to execute a function exactly once on every worker in the sleigh with a fixed set of arguments.

`eachElem` is all that is needed for many basic programs, so that is what we will start with.

First, you need to start up a sleigh, so we'll import the sleigh module, and then construct a Sleigh object:

```
>>> from nws.sleigh import Sleigh
>>> s = Sleigh()
```

This starts three sleigh workers on the local machine, but workers can started on other machines by specifying a launch method and a list of machines.

Let's shut down the sleigh so we can start workers on some other machines.

```
>>> s.stop()
```

This deletes the sleigh's NWS workspace, and shuts down all of the sleigh worker processes.

Now we'll make a new sleigh, starting workers on node1 and node2 using the ssh command, and we'll use an NWS server that's running on node10:

```
>>> from nws.sleigh import sshcmd
>>> s = Sleigh(launch=sshcmd, nodeList=['node1', 'node2'],
...            nwsHost='node10')
```

If you're just starting to use NWS, the first method is very simple and convenient, and you don't have to worry about configuring ssh. Once you've got your NWS program written, and want to run it a network, you just change the way that you construct your Sleigh, and you're ready to run.

If you're running on a machine that has multiple processors, then you can use the simpler method of constructing your Sleigh, but you'll probably want to control the number of workers that get started. To do that, you use the `workerCount` option:

```
>>> s = Sleigh(workerCount=8)
```

There are more options, but that's more than enough to get you started.

So now that we know how to create a sleigh, let actually run a parallel program. Here's how we do it:

```
> result = s.eachElem(abs, range(-10, 0))
```

In that simple command, we have defined a set of data that is processed by multiple workers in parallel, and returned each of the results in a list. (Of course, you would never really bother to do such a trivial amount of work with a parallel program, but you get the idea.)

This `eachElem` command puts 10 tasks into the sleigh workspace. Each task contains one value from -10 to -1. This value is passed as the argument to the absolute value function. The return value of the function is put into the sleigh workspace. The `eachElem` command waits for all of the results to be put into the workspace, and returns them as a list, which are the numbers from 10 to 1.

As a second example, let's add two lists together. First, we'll define an add2 function, and then we'll use it with `eachElem`:

```
>>> def add2(x, y): return x + y
...
>>> result = s.eachElem(add2, [range(10), range(10, 0, -1)])
```

This is the parallel equivalent to the Python command:

```
>>> result = map(add2, range(10), range(10, 0, -1))
```

We can keep adding more list arguments in this way, but there is also a way to add arguments that are the same for every task, which we call *fixed* arguments:

```
>>> result = s.eachElem(add2, range(10), 20)
```

This is equivalent to the Python command:

```
>>> result = map(add2, range(10), [20]*10)
```

The order of the arguments passed to the function are normally in the specified order, which means that the fixed arguments always come after the varying arguments. To change this order, a permutation list can be specified. The permutation list is specified using the `argPermute` keyword parameter.

For example, to perform the parallel equivalent of the Python operation `map(sub, [20]*20, range(20))`, we do the following:

```
>>> def sub(x, y): return x - y
...
>>> result = s.eachElem(sub, range(20), 20, argPermute=[1,0])
```

This permutation list says to first use the second argument, and then use the first, thus reversing the order of the two arguments.

There is another keyword argument, called `blocking`, which, if set to 0, will make `eachElem` return immediately after submitting the tasks, thus making it non-blocking. A `pending` object is returned, which can be used periodically to check how many of the tasks are complete, and also to wait until all tasks are finished. Here's a quick

example:

```
>>> p = s.eachElem(add2, [range(20), range(20, 0, -1)], blocking=0)
>>> while p.check() > 0:
...     # Do something useful for a little while
...     pass
...
>>> result = p.wait()
```

There is also a keyword argument called `loadFactor` that limits the number of tasks that are put into the workspace at the same time. That could be important if you're executing millions of tasks. Setting the load factor to 3 limits the number of tasks in the workspace to 3 times the number of workers in the sleigh. Here's how to do it:

```
>>> result = s.eachElem(add2, [range(1000), range(1000)], loadFactor=3)
```

The results are exactly the same as not using a load factor. Setting this option only changes the way that tasks are submitted by the `eachElem` command.

As you can see, Sleigh makes it easy to write simple parallel programs. But you're not limited to simple programs. You can use NWS operations to communicate between the worker processes, allowing you to write message passing parallel programs much more easily than using MPI or PVM, for example.

See the examples directory for more ideas on how to use Sleigh.

Appendix A

Reference

A.1 Module nws.client

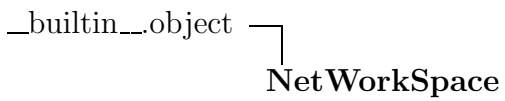
```
Python API for performing NetWorkSpace operations.

NetWorkSpaces (NWS) is a powerful, open-source software package that
makes it easy to use clusters from within scripting languages like
Python, R, and Matlab.  It uses a Space-based approach, similar to
JavaSpaces (TM) for example, that makes it easier to write distributed
applications.

Example:

First start up the NWS server, using the twistd command:

    % twistd -y /etc/nws.tac

Now you can perform operations against it using this module:

    % python
    >>> from nws.client import NetWorkSpace
    >>> nws = NetWorkSpace("test")
    >>> nws.store("answer", 42)
    >>> count = nws.fetch("answer")
```

```
>>> print "The answer is", count
```

A.1.1 Variables

Name	Description
DICT	**Value:** `'dict'` *(type=**str**)*
FIFO	**Value:** `'fifo'` *(type=**str**)*
LIFO	**Value:** `'lifo'` *(type=**str**)*
MULTI	**Value:** `'multi'` *(type=**str**)*
SINGLE	**Value:** `'single'` *(type=**str**)*
STRING	**Value:** `'string'` *(type=**str**)*
V_FETCHERS	**Value:** `2` *(type=**int**)*
V_FINDERS	**Value:** `3` *(type=**int**)*
V_MODE	**Value:** `4` *(type=**int**)*
V_VALUES	**Value:** `1` *(type=**int**)*
V_VARIABLE	**Value:** `0` *(type=**int**)*
WS_MINE	**Value:** `1` *(type=**int**)*
WS_NAME	**Value:** `0` *(type=**int**)*
WS_NUMVARS	**Value:** `4` *(type=**int**)*
WS_OWNER	**Value:** `2` *(type=**int**)*
WS_PERSISTENT	**Value:** `3` *(type=**int**)*
WS_VARLIST	**Value:** `5` *(type=**int**)*

A.1.2 Class NetWorkSpace



Perform operations against workspaces on NWS servers.

The NetWorkSpace object is the basic object used to perform operatations on workspaces. Variables can be declared, created, deleted, and the values of those variables can be manipulated. You can think of a workspace as a network accessible python dictionary, where the variable names are keys in the dictionary, and the associated values are lists of pickled python objects. The store method puts a value into the list associated with the specified variable. The find method returns a single value from a

list. Which value it returns depends on the "mode" of the variable (see the declare method for more information on the variable mode). If the list is empty, the find method will not return until a value is stored in that list. The findTry method works like the find method, but doesn't wait, returning a default value instead (somewhat like the dictionary's get method). The fetch method works like the find method, but it will also remove the value from the list. If multiple clients are all blocked on a fetch operation, and a value is stored into that variable, the server guarantees that only one client will be able to fetch that value. The fetchTry method, not surprisingly, works like the fetch method, but doesn't wait, returning a default value instead.

Methods

__init__(*self*, *wsName*='__default', *serverHost*='localhost', *serverPort*=8765, *useUse*=False, *server*=None, ***opt*)

```
Construct a NetWorkSpace object for the specified NwsServer.

Arguments:

wsName -- Name of the workspace.  There can only be one
        workspace on the server with a given name, so two
        clients can easily communicate with each other by both
        creating a NetWorkSpace object with the same name on the
        same server.  The first client that creates a workspace
        that is willing to take ownership of it, will become the
        owner (see the description of the useUse argument below
        for more information on workspace ownership).

serverHost -- Host name of the NWS server.  This argument is
        ignored if the server argument is not None.  The default
        value is 'localhost'.

serverPort -- Port of the NWS server.  This argument is ignored
        if the server argument is not None.  The default value
        is 8765.
```

```
useUse -- Boolean value indicating whether you only want to use
        this workspace, or if you want to open it (which means
        you're willing to take ownership of it, if it's not
        already owned).

        The default value is False, meaning you are willing to
        take ownership of this workspace.

server -- NwsServer object to associate with this object.  If
        the value is None (the default value), then a NwsServer
        object will be constructed, using the host and port
        specified with the serverHost and serverPort arguments.

        The default value is None.

Keyword Arguments:

persistent -- Boolean value indicating whether the workspace
        should be persistent or not.  If a workspace is
        persistent, it won't be purged when the owner
        disconnects from the NWS server.  Note that only the
        client who actually takes ownership of the workspace
        can make the workspace persistent.  The persistent
        argument is effectively ignored if useUse is True, since
        that client never becomes the owner of the workspace.
        If useUse is False, it is the client who actually
        becomes the owner that determines whether it is
        persistent or not.  That is, after the workspace is
        owned, the persistent argument is ignored.  The default
        value is false.

create -- Boolean value indicating whether the workspace should
        be created if it doesn't already exist.  The default
        value is true.
```

Overrides: __builtin__.object.__init__

__str__(*self*)

Overrides: __builtin__.object.__str__

currentWs(*self*)

Return the name of the current workspace.

ws.currentWs() -> string

declare(*self*, *varName*, *mode*)

```
Declare a variable in a workspace with the specified mode.

ws.declare(varName, mode)

This method is used to specify a mode other than the default
mode of 'fifo'.  Legal values for the mode are:

    'fifo', 'lifo', 'multi', and 'single'

In the first three modes, multiple value can be stored in
a variable.  If the mode is 'fifo', then values are retrieved
in a "first-in, first-out" fashion.  That is, the first value
stored, will be the first value fetched.  If the mode is 'lifo',
then values are retrieved in a "last-in, first-out" fashion,
as in a stack.  If the mode is 'multi', then the order of
retrieval is pseudorandom.

The 'single' mode means that only a single value can be
stored in the variable.  Each new store operation will overwrite
the current value of the variable.

If a variable is created using a store operation, then the
mode defaults to 'fifo'.  The mode cannot be changed with
subsequent calls to declare, regardless of whether the variable
was originally created using store or declare.

Arguments:
```

```
varName -- Name of the workspace variable to declare.

mode -- Mode of the variable.
```

deleteVar(*self*, *varName*)

Delete a variable from a workspace.

ws.deleteVar(varName)

All values of the variable are destroyed, and all currently blocking fetch and find operations will be aborted.

Arguments:

varName – Name of the workspace variable to delete.

fetch(*self*, *varName*)

Return and remove a value of a variable from a workspace.

ws.fetch(varName) -> object

If the variable has no values, the operation will not return until it does. In other words, this is a "blocking" operation. fetchTry is the "non-blocking" version of this method.

Note that if many clients are all trying to fetch from the same variable, only one client can fetch a given value. Other clients may have previously seen that value using the find or findTry method, but only one client can ever fetch or fetchTry a given value.

Arguments:

varName – Name of the workspace variable to fetch.

fetchFile(*self*, *varName*, *fobj*)

Return and remove a value of a variable from a workspace.

ws.fetchFile(varName, fobj) -> number of bytes written to file

Arguments:

varName – Name of the workspace variable to fetch.

fobj – File to write data to.

fetchTry(*self*, *varName*, *missing*=`None`)

Try to return and remove a value of a variable from a workspace.

ws.fetchTry(varName[, missing]) -> object

If the variable has no values, the operation will return the value specified by "missing", which defaults to None.

Note that if many clients are all trying to fetchTry from the same variable, only one client can fetchTry a given value. Other clients may have previously seen that value using the find or findTry method, but only one client can ever fetch or fetchTry a given value.

Arguments:

varName – Name of the workspace variable to fetch.

missing – Value to return if the variable has no values.

fetchTryFile(*self*, *varName*, *fobj*)

Try to return and remove a value of a variable from a workspace.

ws.fetchTryFile(varName, fobj) -> number of bytes written to file

Arguments:

varName – Name of the workspace variable to fetch.

fobj – File to write data to.

find(*self*, *varName*)

Return a value of a variable from a workspace.

ws.find(varName) -> object

If the variable has no values, the operation will not return until it does. In other words, this is a "blocking" operation. findTry is the "non-blocking" version of this method.

Note that (unlike fetch) find does not remove the value. The value remains in the variable.

Arguments:

varName – Name of the workspace variable to find.

findFile(*self*, *varName*, *fobj*)

Return a value of a variable from a workspace.

ws.findFile(varName, fobj) -> number of bytes written to file

Arguments:

varName – Name of the workspace variable to find.

fobj – File to write data to.

findTry(*self*, *varName*, *missing*=`None`)

```
Try to return a value of a variable in the workspace.

ws.findTry(varName[, missing]) -> object

If the variable has no values, the operation will return
the value specified by "missing", which defaults to the value
"None".

Note that (unlike fetchTry) findTry does not remove the value.
The value remains in the variable.

Arguments:

varName -- Name of the workspace variable to use.

missing -- Value to return if the variable has no values.  The
        default is None.
```

findTryFile(*self*, *varName*, *fobj*)

Try to return a value of a variable in the workspace.

ws.findTryFile(varName, fobj) -> number of bytes written to file

Arguments:

varName – Name of the workspace variable to use.

fobj – File to write data to.

ifetch(*self*, *varName*)

Return a fetch iterator for a workspace variable.

ws.ifetch(varName) -> iterator

Unlike ifind, this method doesn't really provide any extra functionality over the fetch method. It is provided for completeness, and for those who just like iterators.

Note that ifetch can be used on FIFO and SINGLE mode variables, but not LIFO and MULTI mode variables.

Arguments:

varName – Name of the workspace variable to iterator over.

ifetchTry(*self*, *varName*)

Return a fetchTry iterator for a workspace variable.

ws.ifetchTry(varName) -> iterator

Unlike ifindTry, this method doesn't really provide any extra functionality over the fetchTry method. It is provided for completeness, and for those who just like iterators.

Note that ifetchTry can be used on FIFO and SINGLE mode variables, but not LIFO and MULTI mode variables.

Arguments:

varName – Name of the workspace variable to iterator over.

ifind(*self*, *varName*)

Return a find iterator for a workspace variable.

ws.ifind(varName) -> iterator

This is very useful if you want to see every value in a variable without destroying them. Unlike the find method, ifind won't return the same value repeatedly. When there are no more values to return, the iterator will block, waiting for a new value to arrive.

Note that ifind can be used on FIFO and SINGLE mode variables, but not LIFO and MULTI mode variables.

Arguments:

varName – Name of the workspace variable to iterate over.

ifindTry(*self*, *varName*)

Return a findTry iterator for a workspace variable.

ws.ifindTry(varName) -> iterator

This is very useful if you want to see every value in a variable without destroying them. Unlike the findTry method, ifindTry won't return the same value repeatedly. When there are no more values to return, the iterator finishes.

Note that ifindTry can be used on FIFO and SINGLE mode variables, but not LIFO and MULTI mode variables.

Arguments:

varName – Name of the workspace variable to iterate over.

listVars(*self*, *wsName*=`None`, *format*=`'string'`)

```
Return a listing of the variables in the workspace.

ws.listVars([wsName[, format]]) -> string or dictionary

Arguments:
```

```
wsName -- Name of the workspace to list.  The default is
        None, which means to use the current workspace.

format -- Output format to return.  Legal values include
        'string' and 'dict'.  The 'string' format returns
        a string which is suitable for printing.
        The 'dict' format returns a dictionary of tuples, where
        the first field is the variable name, the second is
        the number of values, the third is the number of
        fetchers, the fourth is the number of finders, and
        the fifth is the variables mode.  The default value
        is 'string'.
```

store(*self*, *varName*, *val*)

Store a new value into a variable in the workspace.

ws.store(varName, val)

Arguments:

varName – Name of the workspace variable.

val – Value to store in the variable.

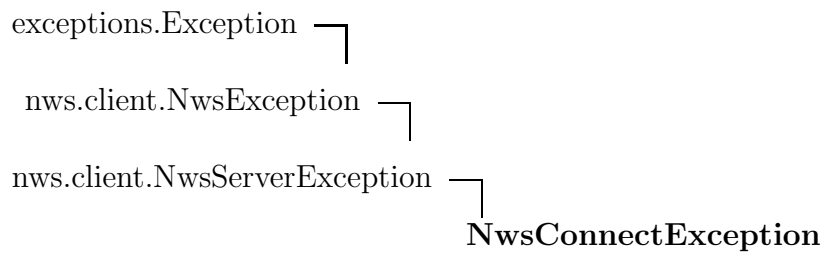
storeFile(*self*, *varName*, *fobj*, *n*=0)

```
Store a new value into a variable in the workspace from a file.

ws.storeFile(varName, fobj[, n]) -> number of bytes read from file

Note that if there is no more data to read from the file,
storeFile returns 0, and does not store a value into the
workspace variable.

Arguments:

varName -- Name of the workspace variable.

fobj -- File to read data from.
```

```
n -- Maximum number of bytes to read from the file.  A value of
        zero means to read and store all of the data in the
        file.  The default value is zero.
```

Inherited from object: __delattr__, __getattribute__, __hash__, __new__, __reduce__, __reduce_ex__, __repr__, __setattr__

A.1.3 Class NwsConnectException

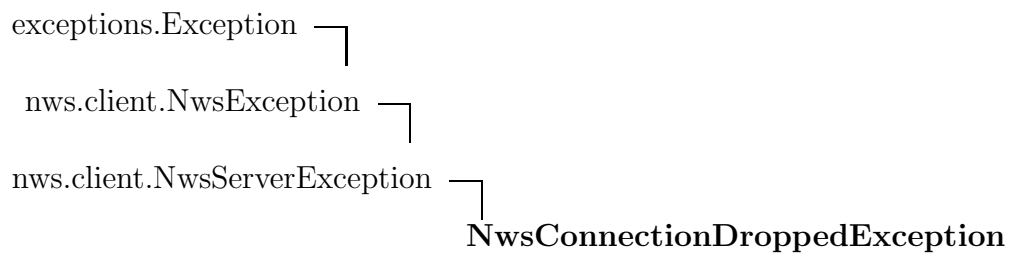


Unable to connect to the NWS server.

Methods

Inherited from Exception: __init__, __getitem__, __str__

A.1.4 Class NwsConnectionDroppedException

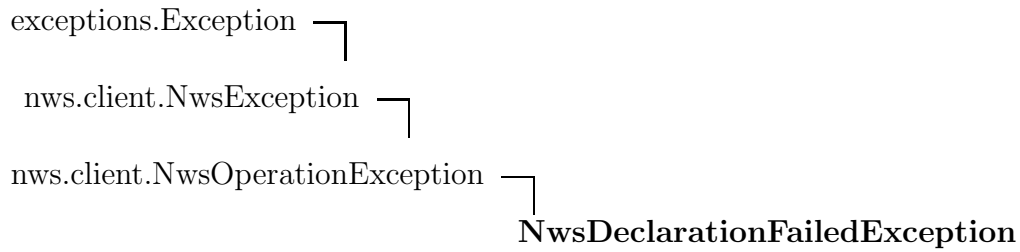


NWS server connection dropped.

Methods

Inherited from Exception: __init__, __getitem__, __str__

A.1.5 Class NwsDeclarationFailedException

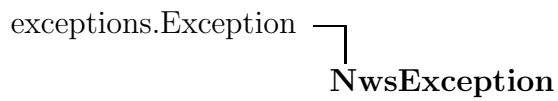


Variable declaration failed.

Methods

Inherited from Exception: __init__, __getitem__, __str__

A.1.6 Class NwsException



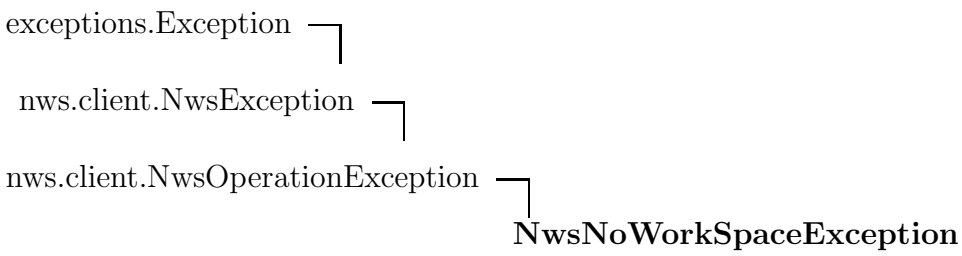
Known Subclasses: NwsOperationException, NwsServerException

Base class for all exceptions raised by this module.

Methods

Inherited from Exception: __init__, __getitem__, __str__

A.1.7 Class NwsNoWorkSpaceException

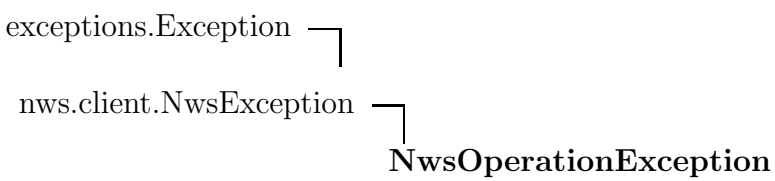


No such workspace.

Methods

Inherited from Exception: __init__, __getitem__, __str__

A.1.8 Class NwsOperationException



Known Subclasses: NwsDeclarationFailedException, NwsNoWorkSpaceException

Error performing an NWS operation.

Methods

Inherited from Exception: __init__, __getitem__, __str__

A.1.9 Class NwsServer

__builtin__.object
└ **NwsServer**

Perform operations against the NWS server.

Operations against workspaces are performed by using the NetWorkSpace class.

Methods

__init__(*self*, *host*='localhost', *port*=8765)

Create a connection to the NWS server.

This constructor is intended for internal use only.

Arguments:

host – Host name of the NWS server.

port – Port of the NWS server.

Overrides: __builtin__.object.__init__

__str__(*self*)

Overrides: __builtin__.object.__str__

close(*self*)

Close the connection to the NWS server.

s.close()

This will indirectly cause the NWS server to purge all non-persistent workspaces owned by this client. Purging may not happen immediately, though, but will depend on the load on the server.

deleteWs(*self*, *wsName*)

Delete the specfied workspace on the NWS server.

s.deleteWs(wsName)

Arguments:

wsName – Name of the workspace to delete.

listWss(*self*, *format*='string')

Return a listing of all of the workspaces on the NWS server.

s.listWss([format]) -> string or dictionary

The listing is a string, consisting of lines, each ending with a newline. Each line consists of tab separated fields. The first field is the workspace name, prefixed with either a '>' or a space, indicating whether the client owns that workspace or not.

mktempWs(*self*, *wsName*='__pyws__%d')

```
Make a temporary workspace on the NWS server.

s.mktempWs([wsName]) -> string

The workspace is named using the template and a number
generated by the server that makes the workspace name unique.

Note that the workspace will be created, but it will not be
owned until some client that is willing to take ownership of it
opens it.

The return value is the actual name of workspace that was
created.

Arguments:

wsName -- Template for the workspace name.  This must be a legal
        'format' string, containing only an integer format
```

```
        specifier.  The default is '__pyws__%d'.

Examples:

    Let's create an NwsServer, call mktempWs to make a
    workspace, and then use openWs to create a NetWorkSpace
    object for that workspace:

        >>> from nws.client import NwsServer
        >>> server = NwsServer()
        >>> name = server.mktempWs('example_%d')
        >>> workspace = server.openWs(name)
```

openWs(*self*, *wsName*, *space*=None, ***opt*)

```
Open a workspace.

s.openWs(wsName[, space]) -> space

If called without a space argument, this method will construct a
NetWorkSpace object that will be associated with this NwsServer
object, and then perform an open operation with it against the NWS
server.  The open operation tells the NWS server that this
client wants to use that workspace, and is willing to take
ownership of it, if it doesn't already exist.

The space argument is only intended to be used from the
NetWorkSpace constructor.

The return value is the constructed NetWorkSpace object.

Arguments:

wsName -- Name of the workspace to open.  If the space argument
        is not None, this value must match the space's name.

space -- NetWorkSpace object to use for the open operation.
        If the value is None, then openWs will construct a
```

```
        NetWorkSpace object, specifying this NwsServer object as
        the space's server.  Note that this argument is only
        intended to be used from the NetWorkSpace constructor.
        The default value is None.

Keyword Arguments:

persistent -- Boolean value indicating whether the workspace
        should be persistent or not.  See the description of the
        persistent argument in the __init__ method of the
        NetWorkSpace class for more information.

create -- Boolean value indicating whether the workspace should
        be created if it doesn't already exist.  The default
        value is true.

Examples:

    Let's create an NwsServer, and then use openWs to create an
    NetWorkSpace object for a workspace called 'foo':

        >>> from nws.client import NwsServer
        >>> server = NwsServer()
        >>> workspace = server.openWs('foo')

    Note that this is (nearly) equivalent to:

        >>> from nws.client import NetWorkSpace
        >>> workspace = NetWorkSpace('foo')
```

useWs(*self*, *wsName*, *space*=None, ***opt*)

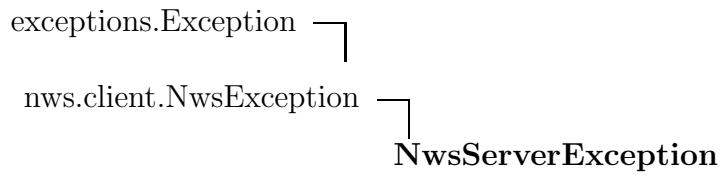
```
Use a NetWorkSpace object.

s.useWs(wsName[, space]) -> space

If called without a space argument, this method will construct a
NetWorkSpace object that will be associated with this NwsServer
```

```
object, and then perform a use operation with it against the NWS
server.  The use operation tells the NWS server that this client
wants to use that workspace, but is not willing to take
ownership of it.

The space argument is only intended to be used from the
NetWorkSpace constructor.

The return value is the constructed NetWorkSpace object.

Arguments:

wsName -- Name of the workspace to use.  If the space argument
        is not None, this value must match the space's name.

space -- NetWorkSpace object to use for the use operation.
        If the value is None, then useWs will construct a
        NetWorkSpace object, specifying this NwsServer object as
        the space's server.  Note that this argument is only
        intended to be used from the NetWorkSpace constructor.
        The default value is None.

Keyword Arguments:

create -- Boolean value indicating whether the workspace should
        be created if it doesn't already exist.  The default
        value is true.

Examples:

    Let's create an NwsServer, and then use useWs to create an
    NetWorkSpace object for a workspace called 'foo':

        >>> from nws.client import NwsServer
        >>> server = NwsServer()
        >>> workspace = server.useWs('foo')

    Note that this is (nearly) equivalent to:
```

```
>>> from nws.client import NetWorkSpace
>>> workspace = NetWorkSpace('foo', useUse=True)
```

Inherited from object: __delattr__, __getattribute__, __hash__, __new__, __reduce__, __reduce_ex__, __repr__, __setattr__

A.1.10 Class NwsServerException



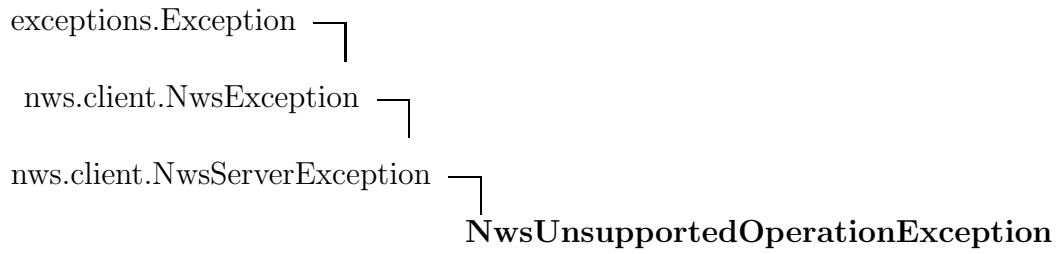
Known Subclasses: NwsConnectException, NwsConnectionDroppedException, NwsUnsupportedOperationException

Error communicating with the NWS server.

Methods

Inherited from Exception: __init__, __getitem__, __str__

A.1.11 Class NwsUnsupportedOperationException



NWS server does not support this operation.

Methods

Inherited from Exception: __init__, __getitem__, __str__

A.1.12 Class NwsValueIterator

__builtin__.object
└ **NwsValueIterator**

Implements the iterated operations against a workspace variable.

Instances of this class are returned from the NetWorkSpace ifetch, ifetchTry, ifind, and ifindTry methods.

Methods

__init__(*self*, *ws*, *varName*, *op*)

Create an iterator over a workspace varible.

This constructor is intended for internal use only.

Arguments:

ws – NetWorkSpace object containing the specified variable.

varName – Name of the workspace variable to iterate over.

op – Operation to perform.

Overrides: __builtin__.object.__init__

__iter__(*self*)

next(*self*)

reset(*self*)

Reset the iterator to the beginning of the workspace variable.

This conveniently restores the state of the iterator to when it was first created.

restart(*self*)

Allow the iterator to continue where it left off after stopping.

The Python iterator protocol requires that iterators continue to raise StopIteration once they've raised it. The NwsValueIterator will do that unless and until you call this method. That can be useful for the "Try" iterators, where you might want to find all of the values in a variable, and at a later point see if there are any new values without having to see the previous ones again.

writeTo(*self*, *fobj*)

Write the next value to the specified file or file-like object.

it.writeTo(fobj) -> number of bytes written to file

This is very much like the "next" method, but unlike "next", is intended to be called explicitly. It provides the same kind of functionality as the various NetWorkSpace findFile/fetchTryFile methods. It is the easiest and most memory efficient way to non-destructively write all of the values of a variable to a file.

Arguments:

fobj – File to write data to.

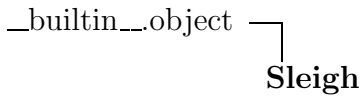
Inherited from object: __delattr__, __getattribute__, __hash__, __new__, __reduce__, __reduce_ex__, __repr__, __setattr__, __str__

A.2 Module nws.sleigh

```
Python API for parallel programming using NetWorkSpaces.

The sleigh module, built on top of NetWorkSpaces (NWS), makes it very
easy to write simple parallel programs.  It contains the Sleigh class,
which provides two basic methods for executing tasks in parallel:
eachElem and eachWorker.
```

```
eachElem is used to execute a specified function multiple times in
parallel with a varying set of arguments.  eachWorker is used to execute
a function exactly once on every worker in the sleigh with a fixed set
of arguments.

Example:

First start up the NWS server, using the twistd command:

    % twistd -y /etc/nws.tac

Now you can create a sleigh to execute python code in parallel:

    % python
    >>> from nws.sleigh import Sleigh
    >>> s = Sleigh()
    >>> import math
    >>> result = s.eachElem(math.exp, range(10))
    >>> print "The answer is", result
```

A.2.1 Class Sleigh



Represents a collection of python processes used to execute tasks.

The sleigh allows python functions, methods, and expressions to be executed in parallel using the eachElem and eachWorker methods.

The sleigh workers are started when the Sleigh object is constructed. When tasks are submitted to the sleigh, using the eachWorker and eachElem methods, the workers will execute the tasks, and return the results. When the stop method is called, the workers are stopped.

Note that a given python program can create multiple Sleigh objects, which will each have it's own set of workers. This could be useful if tasks have different

requirements. For example, you could create a Linux sleigh and a Windows sleigh, and submit Excel tasks only to your Windows sleigh.

Methods

__init__(*self*, **deprecated*, ***kw*)

```
Start the remote python processes used to execute tasks.

Keyword arguments:

launch -- Specifies the method of starting workers.  If this
        argument is set to the string 'local', then the workers
        are executed on the local machine.  If it is set to the
        string 'web', then web launching is used.  Otherwise,
        the launch argument must specify a function (such as
        nws.sleigh.sshcmd) that returns a list of command
        arguments use to execute the workers on the machines
        specified by the nodeList argument.  The default value
        is 'local'.

workerCount -- Number of workers to start if the launch argument
        is set to 'local' (which is the default value of launch).
        This argument is ignored if launch is not set to
        'local'.  The default value is 3.

nodeList -- List of hosts on which to execute workers, if the
        launch argument is set to a function.  This argument is
        ignored if launch is set to 'local' or 'web'.  The
        default value is ['localhost', 'localhost',
        'localhost'].

nwsHost -- Host name of the machine where the NWS server is
        executing.

nwsPort -- Port to connect to the NWS server.
```

```
nwsHostRemote -- Host name of the machine that workers should
        use to connect to the NWS server.  This is useful in
        conjunction with the sshforwardcmd function (see the
        description of scriptExec).  The default is the value of
        the nwsHost argument.

nwsPortRemote -- Port that workers should use to connect to the
        NWS server.  This is useful in conjunction with the
        sshforwardcmd function (see the description of
        scriptExec).  The default is the value of the nwsPort
        argument.

scriptExec -- Python function returning a list of command
        arguments to execute the worker script.  This list of
        command arguments is appended to the list returned by
        the launch function.  The default value is the envcmd
        function (defined in this module), which uses the
        standard 'env' command to execute the script with the
        appropriate environment for the worker.

scriptDir -- Directory on the worker that contains the execution
        script.

scriptName -- Name of the script that executes the worker on the
        remote machines.  This defaults to
        PythonNWSSleighWorker.sh on Unix, and
        PythonNWSSleighWorker.py on Windows.

modulePath -- Directory path to add to sys.path on workers.
        This is often useful for giving the workers access to
        python modules that define the python function to be
        executed by eachWorker or eachElem.  The default value
        is None.

workingDir -- Directory path to use as the current working
        directory for the workers.  The default value is the
        current working directory of the sleigh master.
```

```
logDir -- Directory in which to create the worker log files.
        The default value is None, which leaves the decision
        to the sleigh worker scripts, which generally uses the
        remote system's temporary directory.

user -- User name to use for remote execution of worker.  This
        argument may be ignored, depending the specified
        launch function.  The default is the value of the USER
        environment variable, or the USERNAME environment
        variable if USER is not set.

wsNameTemplate -- Template for the sleigh workspace name.  This
        must be a legal 'format' string, containing only an
        integer format specifier.  The default is
        'sleigh_ride_%04d'.

userWsNameTemplate -- Template for the user's workspace name.
        This must be a legal 'format' string, containing only
        an integer format specifier.  The default is
        'sleigh_user_%04d'.

verbose -- Boolean flag used for displaying debug messages.
        Debug messages will be sent to stderr.  This will also
        cause the worker processes to write debug messages to
        files prefixed with 'sleigh_ride' in their current
        working directory (as controled by the workerDir
        argument).  The default value is False.
```

Overrides: __builtin__.object.__init__

__str__(*self*)

Overrides: __builtin__.object.__str__

eachElem(*self*, *fun*, *elementArgs*=[[]], *fixedArgs*=[], ***kw*)

```
Execute a function, method, or expression for each element in
the specified list(s).
```

```
s.eachElem(fun, elementArgs[, fixedArgs]) -> list or SleighPending

The results are normally returned as a list, unless the blocking
arguments is set to False, in which case, a SleighPending object
is returned.

Arguments:

fun -- Function, method, or python expression to execute.  To
        execute a method, you must specify a bound method
        object.  If the function or defining class is defined in
        a module, the workers will attempt to import that
        module.  If that module isn't available to the worker
        (because it's a non-standard module, not in the
        PYTHONPATH), then the worker is not be able to execute
        those tasks.

        To execute a python expression, you must specify it as a
        string.  Leading whitespace is automatically stripped to
        avoid a common source of python syntax errors.

elementArgs -- List of arguments to pass to the function or method
        that need to be different for different tasks.  In
        general, this is a list of iterable objects, such as lists,
        each containing the values to use for a given argument of
        the different tasks.

        If your function needs only one varying argument of a
        simple type, you can specify it without the outer list.

        Note that for a python expression, the list of arguments
        is passed to the expression as a global variable named
        'SleighArguments'.

fixedArgs -- List of additional arguments that are
        fixed/constant for each task.  Normally, they are
        appended to the arguments specified by elementArgs, but
        the order can be altered using the argPermute argument
```

```
        described below.

        The default value is an empty list, which means that no
        extra arguments are passed to the function.

        Note that for a python expression, the list of arguments
        is passed to the expression as a global variable named
        'SleighArguments'.

Keyword arguments:

type -- Indicates the type of function invocation to perform.
        This can be either 'invoke', 'define', or 'eval'.
        If the fun argument is a function or bound method, then
        the default value is 'invoke'.  If the fun argument is
        a string, then the default value is 'eval' (a value of
        'invoke' or 'define' is illegal for python expressions).

blocking -- A boolean value that indicates whether to wait for
        the results, or to return as soon as the tasks have been
        submitted.  If set to False, eachElem will return a
        SleighPending object that is used to monitor the status
        of the tasks, and to eventually retrieve the results.
        You must wait for the results to be complete before
        executing any further tasks on the sleigh, or a
        SleighOccupiedException will be raised.

        If blocking is set to False, then the loadFactor
        argument is disabled and ignored.  Note that it's
        unlikely that you'll need to turn off blocking in
        eachElem.  Non-blocking mode is more useful in
        eachWorker.

        The default value is True.

argPermute -- List that maps the specified arguments to the
        actual arguments of the execution function.  By
        "specified arguments", I mean the items extracted from
```

```
        elementArgs, followed by fixedArgs.  (Note that unless
        you are specifying both elementArgs and fixedArgs, you
        probably don't need to use argPermute.)  The items in
        the argPermute list are used as indexes into the
        "specified arguments" list.  The length of argPermute
        determines the number of arguments passed to the
        execution function, which would normally be the length
        of the specified arguments list, but this is not
        required.  For example, setting argPermute to an empty
        list would cause the execution function to be called
        without any arguments (although elementArgs would still
        be required, and would be used to determine the number
        of tasks to execute).

        The default behaviour is to pass the execution function
        the arguments specified by elementArgs, followed by the
        arguments from fixedArgs, which is equivalent to setting
        argPermute to:

            n = len(elementArgs) + len(fixedArgs)
            argPermute = range(n)

        If you wished to reverse the order of the arguments, you
        could then modify argPermute:

            argPermute.reverse()

        But, more realistically, you need to interleave the
        fixed arguments with the varying arguments.  For
        example, your execution function takes on fixed
        argument, followed by two that vary, you would set
        argPermute to:

            argPermute=[1,2,0]

loadFactor -- Maximum number of tasks per worker to put into the
        workspace at one time.  This can become important if you
        are executing a very large number of tasks.  Setting
```

```
        loadFactor to 3 will probably keep enough tasks
        available in the workspace to keep the workers busy,
        without flooding the workspace and bogging down the NWS
        server.

        The default behaviour is to submit all of the tasks
        to the sleigh workspace immediately.

accumulator -- A function or callable object that will be called
        for each result as they arrive.  The first argument to
        the function is a list of result values, and the second
        argument is a list of indexes, which identifies which task.

        The arguments to the accumulator function are lists since
        in the future, we plan to allow tasks to be "chunked" to
        improve performance of small tasks.

        Note that bound methods can be very useful accumulators.
```

eachWorker(*self*, *fun*, **workerArgs*, ***kw*)

```
Execute a function, method, or expression on each worker of sleigh.

s.eachWorker(fun[, ...]) -> list or SleighPending

The results are normally returned as a list, unless
the blocking arguments is set to False, in which case, a
SleighPending object is returned.

Arguments:

fun -- Function, method, or python expression to execute.  To
        execute a method, you must specify a bound method
        object.  If the function or defining class is defined in
        a module, the workers will attempt to import that
        module.  If that module isn't available to the worker
        (because it's a non-standard module, not in the
        PYTHONPATH), then the worker is not be able to execute
```

```
        those tasks.

        To execute a python expression, you must specify it as a
        string.  Leading whitespace is automatically stripped to
        avoid a common source of python syntax errors.

Optional arguments:

*workerArgs -- Arguments to pass to the function or method.
        Specify whatever arguments the function requires,
        including no arguments.  The exact same set of arguments
        will be used for each worker (unlike eachElem).  For a
        python expression, these arguments are passed to the
        expression as a global variable named 'SleighArguments'.

Keyword arguments:

type -- Indicates the type of function invocation to perform.
        This can be either 'invoke', 'define', or 'eval'.
        If the fun argument is a function or bound method, then
        the default value is 'invoke'.  If the fun argument is
        a string, then the default value is 'eval' (a value of
        'invoke' or 'define' is illegal for python expressions).

blocking -- A boolean value that indicates whether to wait for
        the results, or to return as soon as the tasks have been
        submitted.  If set to False, eachWorker will return a
        SleighPending object that is used to monitor the status
        of the tasks, and to eventually retrieve the results.
        You must wait for the results to be complete before
        executing any further tasks on the sleigh, or a
        SleighOccupiedException will be raised.

        This argument is important if you want the master to be
        able to interact/communicate with the workers, via NWS
        operations, for example.  This allows you to implement
        more complex parallel or distributed programs.
```

```
        The default value is True.

accumulator -- A function or callable object that will be called
        for each result as they arrive.  The first argument to
        the function is a list of result values, and the second
        argument is a list of indexes, which identifies which task.

        The arguments to the accumulator function are lists since
        in the future, we plan to allow tasks to be "chunked" to
        improve performance of small tasks.

        Note that bound methods can be very useful accumulators.
```

imap(*self*, *fun*, **iterables*, ***kw*)

```
Return an iterator whose values are returned from the function
evaluated with a argument tuple taken from the given iterable.
Stops when the shortest of the iterables is exhausted.

s.imap(fun, *iterables[, **kw]) -> SleighResultIterator

This is intended to be very similar to the imap function
in the itertools module.  Other than being a method, rather than
a function, this method takes additional, keyword arguments, and
the iterator that is returned has special methods and properties.
See the SleighResultIterator documentation for more information.

Arguments:

fun -- Function, method, or python expression to execute.  To
        execute a method, you must specify a bound method
        object.  If the function or defining class is defined in
        a module, the workers will attempt to import that
        module.  If that module isn't available to the worker
        (because it's a non-standard module, not in the
        PYTHONPATH), then the worker is not be able to execute
        those tasks.
```

```
        To execute a python expression, you must specify it as a
        string.  Leading whitespace is automatically stripped to
        avoid a common source of python syntax errors.


*iterables -- One or more iterables, one for each argument needed
        by the function.

Keyword arguments:

type -- Indicates the type of function invocation to perform.
        This can be either 'invoke', 'define', or 'eval'.
        If the fun argument is a function or bound method, then
        the default value is 'invoke'.  If the fun argument is
        a string, then the default value is 'eval' (a value of
        'invoke' or 'define' is illegal for python expressions).

loadFactor -- Maximum number of tasks per worker to put into the
        workspace at one time.  This can become important if you
        are executing a very large number of tasks (and essential
        if submitting infinite tasks).  The default value of 10
        will probably keep enough tasks available in the workspace
        to keep the workers busy, without flooding the workspace
        and bogging down the NWS server.
```

starmap(*self*, *fun*, *iterable*, ***kw*)

```
Return an iterator whose values are returned from the function
evaluated with a argument tuple taken from the given iterable.
Stops when the shortest of the iterables is exhausted.

s.starmap(fun, iterable[, **kw]) -> SleighResultIterator

This is intended to be very similar to the starmap function
in the itertools module.  Other than being a method, rather than
a function, this method takes additional, optional arguments, and
the iterator that is returned has special methods and properties.
See the SleighResultIterator documentation for more information.
```

```
Arguments:

fun -- Function, method, or python expression to execute.  To
        execute a method, you must specify a bound method
        object.  If the function or defining class is defined in
        a module, the workers will attempt to import that
        module.  If that module isn't available to the worker
        (because it's a non-standard module, not in the
        PYTHONPATH), then the worker is not be able to execute
        those tasks.

        To execute a python expression, you must specify it as a
        string.  Leading whitespace is automatically stripped to
        avoid a common source of python syntax errors.


iterable -- Returns argument tuples for the function.

Keyword arguments:

type -- Indicates the type of function invocation to perform.
        This can be either 'invoke', 'define', or 'eval'.
        If the fun argument is a function or bound method, then
        the default value is 'invoke'.  If the fun argument is
        a string, then the default value is 'eval' (a value of
        'invoke' or 'define' is illegal for python expressions).

loadFactor -- Maximum number of tasks per worker to put into the
        workspace at one time.  This can become important if you
        are executing a very large number of tasks (and essential
        if submitting infinite tasks).  The default value of 10
        will probably keep enough tasks available in the workspace
        to keep the workers busy, without flooding the workspace
        and bogging down the NWS server.
```

status(*self*, *closeGroup*=`False`, *timeout*=`0.0`)

```
Return the status of the worker group.

s.status(closeGroup, timeout) -> numworkers, closed

The status includes the number of workers that have joined the
group so far, and a flag that indicates whether the group has
been closed (meaning that no more workers can join).  Normally,
the group is automatically closed when all the workers that were
listed in the constructor have joined.  However, this method
allows you to force the group to close after the timeout
expires.  This can be particularly useful if you are running
on a large number of nodes, and some of the nodes are slow or
unreliable.  If some of the workers are never started, the group
will never close, and no tasks will ever execute.

Arguments:

closeGroup -- Boolean flag indicating whether to close the
        group.  If True, the group will be closed, after the
        specified timeout.  The default value is False.

timeout -- Number of seconds to wait for the group to close
        before returning.  The default value is 0.0.
```

stop(*self*)

Stop the remote processes and delete the sleigh workspace.

s.stop()

Inherited from object: __delattr__, __getattribute__, __hash__, __new__, __reduce__, __reduce_ex__, __repr__, __setattr__

A.2.2 Class SleighException

exceptions.Exception
└ **SleighException**

Known Subclasses: SleighGatheredException, SleighIllegalValueException, SleighJoinException, SleighNwsException, SleighOccupiedException, SleighScriptException, SleighStoppedException, SleighTaskException

Base class for all exceptions raised by this module.

Methods

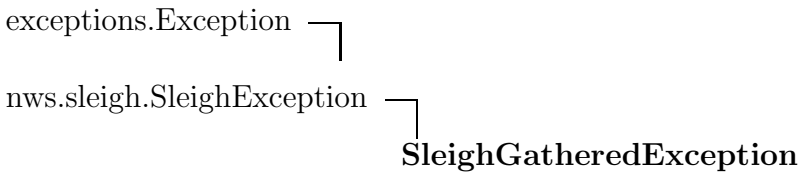
__repr__(*self*)

__str__(*self*)

Overrides: exceptions.Exception.__str__

Inherited from Exception: __init__, __getitem__

A.2.3 Class SleighGatheredException

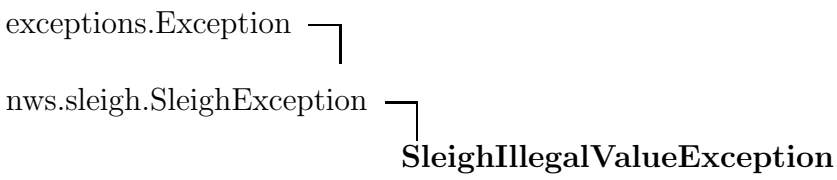


Results already gathered.

Methods

Inherited from Exception: __init__, __getitem__
Inherited from SleighException: __repr__, __str__

A.2.4 Class SleighIllegalValueException

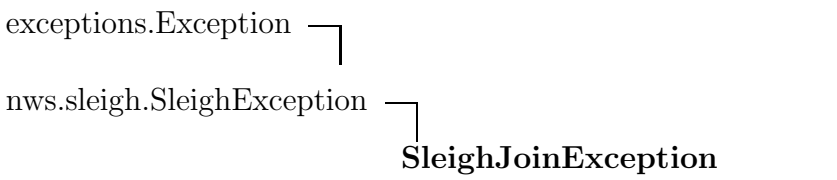


Illegal value specified.

Methods

Inherited from Exception: __init__, __getitem__
Inherited from SleighException: __repr__, __str__

A.2.5 Class SleighJoinException

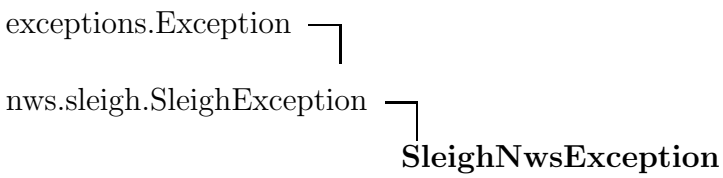


Too late to join worker group.

Methods

Inherited from Exception: __init__, __getitem__
Inherited from SleighException: __repr__, __str__

A.2.6 Class SleighNwsException

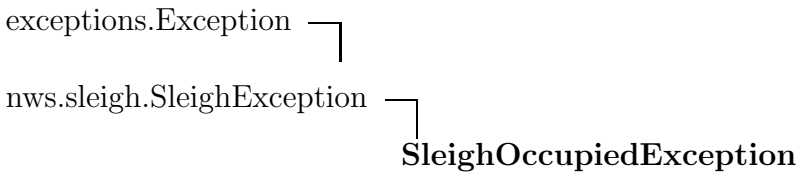


Error performing NWS operation.

Methods

Inherited from Exception: __init__, __getitem__
Inherited from SleighException: __repr__, __str__

A.2.7 Class SleighOccupiedException

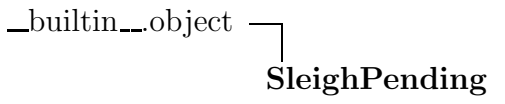


Sleigh is occupied.

Methods

Inherited from Exception: __init__, __getitem__
Inherited from SleighException: __repr__, __str__

A.2.8 Class SleighPending



Represents a sleigh eachWorker/eachElem invocation in progress.

This is returned from the eachElem and eachWorker operations when executed asynchronously. It allows you to check for the number of tasks left to be executed, and to wait for the results of the operation to be returned.

Methods

__init__(*self*, *nws*, *numTasks*, *barrierName*, *sleighState*, *accumulator*)

```
Create an object that represents the pending sleigh operation.

This constructor is intended for internal use only.

Arguments:

nws -- Sleigh NetWorkSpace object.

numTasks -- Number of tasks that were submitted.

barrierName -- Name of the barrier to wait at when complete.

sleighState -- Object representing the current state of the
        sleigh.

accumulator -- Function to call with results as they arrive.
```

Overrides: __builtin__.object.__init__

__str__(*self*)

Overrides: __builtin__.object.__str__

check(*self*)

Return the number of tasks still outstanding.

p.check() -> integer

wait(*self*)

Wait for and return the list of results.

p.wait() -> list

Inherited from object: __delattr__, __getattribute__, __hash__, __new__, __reduce__, __reduce_ex__, __repr__, __setattr__

A.2.9 Class SleighResultIterator

__builtin__.object
└ **SleighResultIterator**

Returns results from tasks submitted to the sleigh.

Instances of this class are returned from the Sleigh imap, and starmap methods.

Methods

__init__(*self*, *task*, *taskIter*, *starIter*, *nws*, *numSubmitted*, *sleighState*)

Create an iterator over the task results.

This constructor is intended for internal use only.

Arguments:

task – Partially initialized Task object.

taskIter – Iterator over the task arguments.

starIter – Is this a "star" iteration?

nws – Sleigh NetWorkSpace object.

numSubmitted – Number of tasks already submitted.

sleighState – Part of the Sleigh objects internal state.

Overrides: __builtin__.object.__init__

__iter__(*self*)

next(*self*)

shutdown(*self*)

Stop submitting tasks from the iterator.

This method is a less drastic version of "stop". It is expected that you will keep retrieving results that have already been submitted, but no new tasks will be submitted, regardless of what tasks were originally specified to imap or starmap. The sleigh object will continue to be "occupied" until all results of the pending tasks have been retreived.

stop(*self*)

Stop the iterator, flushing any pending results.

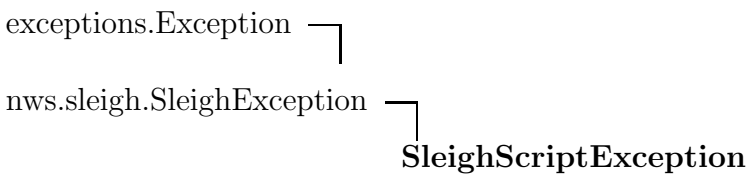
This method is useful if you're done with the iterator, and don't want to retrieve anymore results. After calling stop, you can submit more tasks to the sleigh (that is, it will no longer be "occupied".

Inherited from object: __delattr__, __getattribute__, __hash__, __new__, __reduce__, __reduce_ex__, __repr__, __setattr__, __str__

Properties

Name	Description
buffered	Number of buffered task results.
returned	Number of task results returned.
stopped	Is the iterator stopped?
submitted	Number of submitted tasks.

A.2.10 Class SleighScriptException

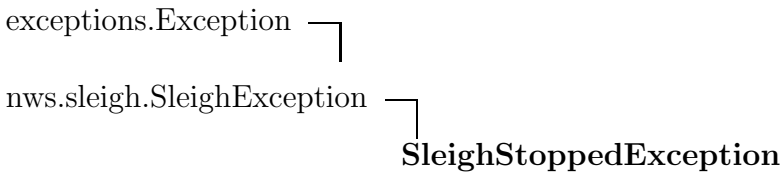


Unable to find sleigh worker script.

Methods

Inherited from Exception: __init__, __getitem__
Inherited from SleighException: __repr__, __str__

A.2.11 Class SleighStoppedException

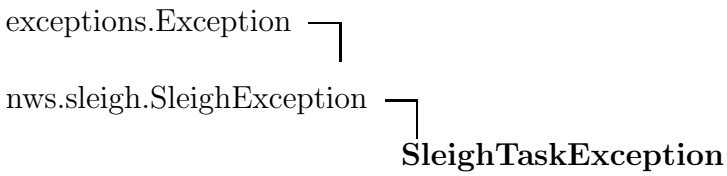


Sleigh is stopped.

Methods

Inherited from Exception: __init__, __getitem__
Inherited from SleighException: __repr__, __str__

A.2.12 Class SleighTaskException



Error executing a task.

Methods

Inherited from Exception: __init__, __getitem__
Inherited from SleighException: __repr__, __str__

Appendix B

Remote Execution Configuration

B.1 Setting Up a Password-less SSH Login

- To generate public and private keys, follow the steps below:
  - Open a terminal (shell on UNIX and DOS prompt on Windows).
  - `ssh-keygen -t rsa`
    (assume ssh-keygen is in your PATH)
  - `cd .ssh`
    (.ssh directory is located in your HOME directory, /home/user on UNIX or Cygwin and C:\Program Files\copssh\home\user on Windows)
  - `cp id_rsa.pub authorized_keys`
    This step allows password-less login to local machine.
  - For all remote machines that you want password-less login, append the content of id_rsa.pub to their authorized_keys file.
- To test the password-less login, type the following command:

  ```
  % ssh hostname date
  ```

  If everything is setup correctly, you should not be asked for password and the current date on remote machine will be returned.