

SCAI

NetWorkSpaces[®] for Python

User Guide

SCIENTIFIC
COMPUTING ASSOCIATES
INC.

One Century Tower
265 Church Street, New Haven, CT 06510-7010
Tel: (203) 777-7442 • Fax (203) 776-4074

www.LindaSpaces.com

Acknowledgements

D. Atapattu, N. Carriero, J. Lai, S. Weston
Scientific Computing Associates, Inc.

© Copyright 2006 Scientific Computing Associates, Inc. All rights reserved. SCAI, NetWorkSpaces, and Sleigh are trademarks of Scientific Computing Associates, Inc. All other trademarks and registered trademarks are property of their respective holders.

Manual version 1.0 January 2006. Printed in the USA.

Contents

| | |
|--|-----------|
| Introduction | 1 |
| Getting Started | 5 |
| Prerequisites | 5 |
| NetWorkSpaces Server..... | 5 |
| Starting the Server..... | 5 |
| Stopping the Server | 6 |
| NetWorkSpaces Client | 6 |
| Installing the Client | 6 |
| Starting the NetWorkSpaces Client | 7 |
| Starting Sleigh..... | 8 |
| Setting Up a Password-less SSH Login | 8 |
| Alternative to SSH..... | 9 |
| Tutorials | 11 |
| Python NetWorkSpaces Tutorial..... | 11 |
| Python Sleigh Tutorial..... | 13 |
| Appendix | 17 |
| NetWorkSpaces for Python API Reference | 17 |
| nws.client | 17 |
| currentWs(self)..... | 19 |
| declare(self, varName, mode)..... | 19 |
| deleteVar(self, varName)..... | 20 |
| fetch(self, varName)..... | 20 |
| fetchTry(self, varName, missing=None)..... | 20 |
| find(self, varName) | 21 |
| findTry(self, varName, missing=None)..... | 21 |
| listVars(self, wsName=None) | 21 |
| store(self, varName, val)..... | 21 |
| class NwsServer | 22 |
| close(self) | 22 |
| deleteWs(self, wsName) | 22 |
| listWss(self)..... | 22 |
| mktempWs(self, wsName='__pyws__%d')..... | 23 |
| openWs(self, wsName, space=None, **opt)..... | 23 |
| useWs(self, wsName, space=None)..... | 24 |
| Data | 25 |

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.

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.

`find()` reads a value without removing it.

Note that `fetch()` and `find()` block if no value is bound to the name. `fetchTry()` and `findTry()` are non-blocking; they return an empty value or 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 `NWSSServer` 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 object is an attribute of the `NetWorkSpace`, and you can access it using the Python syntax `netWorkSpace_object@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.

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()` to invoke a function on each element of a list, with the invocations being evaluated concurrently by the sleigh participants.

`eachWorker()` to have each participant in a sleigh invoke a function. You can use this operation, for example, to build up requisite state prior to an `eachElem()` invocation.

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 Python programs as well.



2

Getting Started

This chapter provides step-by-step procedures for setting up *NetWorkSpaces* for Python.

Prerequisites

NetWorkSpaces for Python runs on Linux, Mac OS/X, and Windows XP. NetWorkSpaces and Sleigh require the following:

- Python 2.3 or later — <http://www.python.org>
We recommend ActiveState Python — <http://www.activestate.com> for Windows.
- An `ssh` client and server are needed by Sleigh (by default)

The NetWorkSpaces server also requires:

- Twisted 2.1 and Twisted-Web 0.5 or later: — <http://twistedmatrix.com>.

NetWorkSpaces Server

Starting the Server

There are two ways to start a NetWorkSpaces server:

- Using the `twistd` command:

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

- Or using the `nws` script on Posix systems:

```
% nws start
```

This starts a NetWorkSpaces server on port `8765` and the web interface on port `8766`.

Stopping the Server

There are also two ways to stop a NetWorkSpaces server:

- If you used the `twistd` command to start the server, then"

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

- If you used the `nws` script to start the server, then

```
% nws stop
```

NetWorkSpaces Client

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 included in the source distribution, but here's a quick summary of the "System Installation":

1. To install the server on Linux and Mac OS/X:

```
% tar xzf nwserver-1.2.tar.gz
% cd nwserver-1.2
% python setup.py install
```

2. To install the client on Linux and Mac OS/X:

```
% tar xzf nwsclient-1.2.tar.gz
% cd nwsclient-1.2
% python setup.py install
```

You'll need to use a privileged account to do this. If you don't want to do that, or can't, you can use some options to install into a different location. To get help on the install options, execute 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.

Starting the 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 value of `1` into the variable named `x` in that workspace.

If you encounter an error importing the NetWorkSpace class, it's likely that you didn't set up PYTHONPATH correctly.

You can also view what's in the workspace using a web interface. To do this, you point your browser to http://server_host_name:8766, where `server_host_name` is the machine that a NetWorkSpaces server resides on.

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 it, without any help from the *babelfish*, but if the value is any other type of Python object, it needs *babelfish*'s help.

The Python *babelfish* is named `pybabelfish`. It 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 can execute the *babelfish* with the command:

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

or, if you're using a `csch`-compatible shell:

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

or, on Windows:

```
C:\> start pybabelfish
```

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

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

```
# pybabelfishd
```

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

See the [Tutorials](#) chapter for more information about using NetWorkSpaces..

Starting Sleigh

Sleigh is a Python module, built on top of the 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. Sleigh uses ssh (by default) to start the workers. Therefore, before we can get started with Sleigh, we need to setup password-less ssh login.

Note: If you use Windows, where SSH is not available, please refer to the section, [Alternative to SSH](#).

Setting Up a Password-less SSH Login

To generate public and private keys:

1. `ssh-keygen -t rsa`
2. `cd ~/.ssh`
3. `cp id_rsa.pub authorized_keys`
4. For all remote machines for which you want a password-less login, append the content of `id_rsa.pub` to their `authorized_keys` file.

To test the password-less login:

```
% ssh host date
```

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

Alternative to SSH

If you're using workers on Windows, web launch is the recommended way of starting workers at the present. It's also useful on Posix systems, because it makes far fewer assumptions of how things are installed. It is more work, however, since it basically involves manually starting each worker.

To use web launch, you need to set the Sleigh launch argument to the string `web`. The constructor won't return until you've started all of the workers that you want to use. To start a worker, you have to log in to each machine, and perform the following steps:

1. Start the Python interpreter.
2. Using a web browser, go to the web interface of the NWS server that is used by the Sleigh master (i.e., <http://nwsmachine:8766>).
3. Click on the workspace of your Sleigh; Sleigh workspaces start with the prefix `'sleigh_ride'`, unless you specified otherwise.
4. Click on the `runMe` variable.
5. Copy the text of one of the values. The text will be something like:

```
import nws.sleigh;  
nws.sleigh.webLaunch('sleigh_ride_004_tmp', 'n1.xyz.com', 8765)
```

6. Paste that text into your Python session, executing it as a Python command.

Note that you can start multiple workers on a single machine, which is useful for SMPs and for testing.

When you've started up all the workers you want, go to the Sleigh workspace with a web browser, and click on the `deleteMeWhenAllWorkersStarted` variable, and then click on the `Remove Variable?` link to delete that variable. This signals to the Sleigh constructor that all the workers have been started, and your Sleigh script can continue executing.



Python NetWorkSpaces Tutorial

NetWorkSpaces (*NWS*) is an Python package that makes it very easy for different Python programs running (potentially) on 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. Next, an NWS server must be started. This can be done from a shell using the `nws` command, as follows:

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

From another window, 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 sets `age` to `17`.

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 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 writing 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`.

Those 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. In the examples above, we took care never to block because we were only using one Python session.

To use two Python session, 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 a 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 **NWS** server's web interface to see what's going on in your workspaces. Just point your web browser to the URL: <http://localhost:8766>. If you're using a browser on a machine other than the **NWS** server, you must use the appropriate hostname in place of `localhost`.

Python Sleigh Tutorial

Sleigh is a python package, built on top of **NWS**, that makes it very easy to write simple 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 most basic programs, so that is what we'll start with.

First, we 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 using `ssh`, but a list argument can be used to specify the machines to start the workers on. If this fails, it's probably because `ssh` isn't properly installed and configured.

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, with workers on `node1` and `node2`, and we'll use an NWS server that's running on `node10`:

```
>>> s = Sleigh(nodeList=['node1', 'node2'], nwsHost='node10')
```

Now we're ready to run a parallel program. Here it is:

```
>>> 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 in which arguments are passed to the function is normally 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`, makes `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 enables watermarking. This limits the number of tasks that are put into the workspace at the same time, which can be important if you're executing a lot of tasks. For example, setting the `loadFactor` 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 `loadFactor`. 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.



NetWorkSpaces for Python API Reference

nws.client

Python API for performing NetWorkSpace operations.

Description

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™ for example, that makes it easier to write distributed applications.

Example

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

```
% twistd -noy /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
>>>
```

Classes

`NetWorkSpace`

`NwsServer`

```
class NetWorkSpace
```

Perform operations against workspaces on NWS servers. The `NetWorkSpace` object is the basic object used to perform operations 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 Defined Here

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

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

Arguments

- | | |
|-------------------------|--|
| <code>wsName</code> | 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 <code>NetWorkspace</code> 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 <code>useUse</code> argument below for more information on workspace ownership). |
| <code>serverHost</code> | Host name of the NWS server. This argument is ignored if the <code>server</code> argument is not <code>None</code> . The default value is <code>localhost</code> . |
| <code>serverPort</code> | Port of the NWS server. This argument is ignored if the <code>server</code> argument is not <code>None</code> . The default value is <code>8765</code> . |
| <code>useUse</code> | 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 <code>False</code> , meaning you are willing to take ownership of this workspace. |
| <code>server</code> | <code>NwsServer</code> object to associate with this object. If the value is <code>None</code> (the default value), then a <code>NwsServer</code> object will be constructed, using the host and port specified with the <code>serverHost</code> and <code>serverPort</code> arguments. The default value is <code>None</code> . |

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.

currentWs(self)

Return the name of the current workspace.

declare(self, varName, mode)

Declare a variable in a workspace with the specified 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 is 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 undefined. 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 variable to declare.

mode Mode of the variable.

deleteVar(self, varName)

Delete a variable from a workspace.

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

Arguments

`varName` Name of the variable to delete.

fetch(self, varName)

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

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 variable to fetch.

fetchTry(self, varName, missing=None)

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

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 variable to fetch.

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

find(self, varName)

Return a value of a variable from a workspace.

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 variable to find.

findTry(self, varName, missing=None)

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

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 variable to use.

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

listVars(self, wsName=None)

Return a listing of the variables in the workspace.

Arguments

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

store(self, varName, val)

Store a new value into a variable in the workspace.

Arguments

`varName` Name of the variable.

`val` Value to store in the variable.

class NwsServer

Perform operations against the NWS server.

Operations against workspaces are performed by using the NetWorkSpace class.

Methods Defined Here

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

Create a connection to the NWS server.

This constructor is only intended to be called internally.

Arguments

`host` Host name of the NWS server.

`port` Port of the NWS server.

close(self)

Close the connection to the NWS server.

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 specified workspace on the NWS server.

Arguments

`wsName` Name of the workspace to delete.

listWss(self)

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

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.

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.

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.

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)

Use a `NetWorkspace` object.

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`.

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)
```

Data

```
FIFO = 'fifo'
LIFO = 'lifo'
MULTI = 'multi'
SINGLE = 'single'
__all__ = ['NwsServer', 'NetWorkspace', 'FIFO', 'LIFO', 'MULTI',
'SING...']
```



A

Appendix 17

G

Getting Started 5

I

Installing the Client 6

Introduction 1

N

NetWorkSpaces Client 6

NetWorkSpaces for Python API Reference 17

class NwsServer 22

close(self) 22

currentWs(self) 19

Data 25

declare(self, varName, mode) 19

deleteVar(self, varName) 20

deleteWs(self, wsName) 22

fetch(self,varName) 20

fetchTry(self, varName, missing=None) 20

find(self, varName) 21

findTry(self, varName, missing=None) 21

listVars(self, wsName=None) 21

listWss(self) 22

mktempWs(self, wsName='__pyws__%d') 23

nws.client 17

openWs(self, wsName, space=None, **opt) 23

store(self, varName, val) 21

useWs(self, wsName, space=None) 24

NetWorkSpaces Server 5

P

Prerequisites 5

Python NetWorkSpaces Tutorial 11

Python Sleigh Tutorial 13

S

Setting Up a Password-less SSH Login 8

Starting Sleigh 8

Starting the NetWorkSpaces Client 7

Starting the Server 5

Stopping the Server 6

T

Tutorials 11

SCIENTIFIC
COMPUTING ASSOCIATES
INC.

One Century Tower
265 Church Street, New Haven, CT 06510-7010
Tel: (203) 777-7442 • Fax (203) 776-4074

www.LindaSpaces.com