pyTOOLS 2.7

Programmers manual

February 5, 2015

Abstract

Python is an interpreted, object-oriented scripting language. This document describes how to use the pyTOOLS extension module that makes it possible to control FAST/TOOLS from the Python scripting language.

You don’t need any previous knowledge of the Python scripting language to be able to read this document. All is explained in this document.

Revision history

Version:	Date:	Changes:
1.0	Nov 30, 2000	- First version documented.
2.0	May 10, 2001	- Added event notification to pyTOOLS-DSS interface. - Enhancements to pyTOOLS exception handling. - Support for composed indexes. - Some textual and layout changes.
2.2	May 30, 2002	- Added BUS/FAST support. - Add chapter about XML and XML-RPC. - Some textual and layout changes.
2.3	June 14, 2003	- Added DSS date and time functions. - Some textual and layout changes.
2.4	September 19, 2006	- Added support for memo-type fields - Add busSnap function - Add item history functions - Add setup file functions - Add FAST/TOOLS standard directory functions
2.5	October 12, 2008	- Add umh message logging - Remove outdated XML chapter - Some textual and layout changes
2.7	February 5, 2015	- Remove outdated BUS/FAST XML-RPC chapter. - Some textual and layout changes.

Table of contents

CHAPTER 1

Getting to know pyTOOLS

The language you speak affects what you can think.
-- Bruce Eckel

In this chapter we will cover the groundwork. It explains what pyTOOLS is and why you should use it.

1.1 What is pyTOOLS ?

The pyTOOLS package lets you write scripts for the popular FAST/TOOLS SCADA environment using the Python programming language. It's a kind of lightweight interface between FAST/TOOLS and the Python programming language.

With the advent of the Data Set Services (DSS) layer in FAST/TOOLS 8.0 it has become possible to approach FAST/TOOLS data thru a consistent and easy to use programmer's interface. This has inspired the author to write an extension module for his favourite programming language.

1.2 What is Python ?

python,	(*Gr. Myth.* An enormous serpent that lurked in the cave of Mount Parnassus and was slain by Apollo) **1.** any of a genus of large, non-poisonous snakes of Asia, Africa and Australia that suffocate their prey to death. **2**. popularly, any large snake that crushes its prey. **3.** totally awesome, very high-level programming language that can now interact with FAST/TOOLS.

Python is an interpreted, interactive, object-oriented programming language. It is often compared to Tcl, Perl, Scheme or Java. Python combines remarkable power with very clear syntax. It has modules, classes, exceptions, very high-level dynamic data types, and dynamic typing. There are interfaces to many system calls and libraries, as well as to various windowing systems (X11, Motif, Tk, Mac, MFC). New built-in modules are easily written in C or C++. Python is also used as an extension language for applications that need a programmable interface.

The Python implementation is portable: it runs on many brands of UNIX, on Windows, OS/2, Mac, Amiga... If your favourite system isn't listed here, it may still be supported, if there's a C compiler for it. Ask around on comp.lang.python -- or just try compiling Python yourself. It's Free!

1.3 What are Python extensions?

Python extensions are used to extend the Python interpreter with functions written in C or C++. To support extensions, the Python API (Application Programmers Interface) defines a set of functions, macros and variables that provide access to most aspects of the Python interpreter. Extension modules are compiled into dynamic link libraries and placed in the Python extension directory. From then on they can be loaded at runtime. The main reasons to write a Python extension is to increase execution speed and to interface with another application via its C-API.

1.4 Why should I use it?

Consider this: *pyTOOLS makes it possible to interact directly with FAST/TOOLS from within a script*. Each FAST/TOOLS project produces a collection of scripts for configuration and maintenance. The currently used scripts offer no direct access to FAST/TOOLS data or any of the FAST/TOOLS C-API functions. Using pyTOOLS you are able to interact directly with FAST/TOOLS from within a script. In many cases this will eliminate the need to write C-code because everything that can be done using the DSS C-API can also be done using a script.

I am convinced there is a need for a powerful, easy to learn scripting language that moves FAST/TOOLS scripting into areas that were up till now reserved for C-programmers.

1.5 The Zen of Python

Although the following list appeared on the Python humour page it sums up nicely what Python is about.

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
*Although never is often better than *right* now.*
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!

- Tim Peters

1.6 How is this document organized?

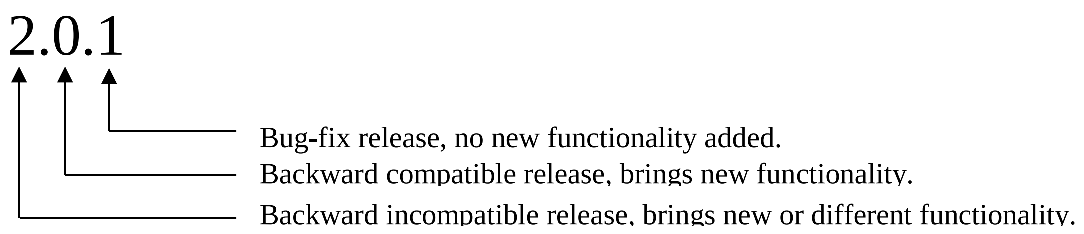
This document is intended to be both an introduction to the pyTOOLS package and a reference manual. It is structured as follows:

- Chapter 1
  This chapter which provides an introduction.
- Chapter 2
  Explains how to install and configure the Python interpreter and the pyTOOLS extension.
- Chapter 3
  Provides a basic explanation of the concepts behind DSS. It will give you all the DSS knowledge you need to successfully use the pyTOOLS DSS module.
- Chapter 4
  Contains an informal introduction to the possibilities of the pyTOOLS DSS module. A small tutorial if you like.
- Chapter 5
  This chapter contains the formal description all the pyTOOLS DSS module functions and data structures.
- Chapter 6
  Explains pyTOOLS exception handling.
- Chapter 7
  Contains a description of the pyTOOLS data structures.
- Chapter 8
  Tells you where to go for further orientation.

1.7 pyTOOLS version numbering:

pyTOOLS release numbers consist of two or (if needed) tree numbers separated by a dot:

Example:


2.0.1
Bug-fix release, no new functionality added.
Backward compatible release, brings new functionality.
Backward incompatible release, brings new or different functionality.

CHAPTER 2

Setup and Installation

Debugging is twice as hard as writing the code in the first place.
Therefore, if you write the code as cleverly as possible,
you are, by definition, not smart enough to debug it.

-- Brian Kernighan

This chapter explains how to install Python for the Windows operating system. Everything takes less than 5 minutes if you have the pyTOOLS cd-rom at hand.

2.1 Availability

The pyTOOLS DSS-module is currently available for the Windows platform and for Linux. The pyTOOLS source code is made available on the FAST/TOOLS installation CD.

2.2 What do I need to install?

Apart from FAST/TOOLS off course, you need to install the Python interpreter and the pyTOOLS extension, which consists of one single file.

2.3 Installing Python

For the Windows platform you can download Python for the website www.python.org. For the current pyTOOLS version you need to download and install Python 2.7

2.4 Installing pyTOOLS

To install pyTOOLS copy the file `dss.pyd` to the Python dll directory. If you choose the default directory during installation this will be: `c:\Python27\DLLs`. The file `dss.pyd` is a dynamic link library (dll) that contains the interface functions between FAST/TOOLS and the Python interpreter. Don’t be confused by the extension .pyd. Windows dynamic link library files can have any extension you like. The .pyd extension is used to uniquely identify Python extension dll files.

The current version of the python interpreter supported by pyTOOLS is Python 2.7.

2.5 pyTOOLS license

2.6 Python interpreter

Python development has been sponsored by a different organization over the years, all which have copyright at some portion of Python. Python can be down loaded from: www.python.org.

All licencing python licencing information can be found at https://docs.python.org/2/license.html

CHAPTER 3

Just enough DSS

Python, It fits your brain
-- Motto of the 9th Annual Python Conference

This section describes the DSS at a global level. It is meant as an introduction to DSS and tells you everything you need to know about DSS to use the pyTOOLS module. Readers familiar with the principles behind DSS can skip this chapter.

3.1 What is DSS?

The DSS or Data Set Services were introduced in FAST/TOOLS to offer a consisted gateway to FAST/TOOLS data. DSS hides individual tool interfaces from the programmer. As figure 3.1 shows DSS forms an abstraction layer between FAST/TOOLS processes and the applications. Applications that use

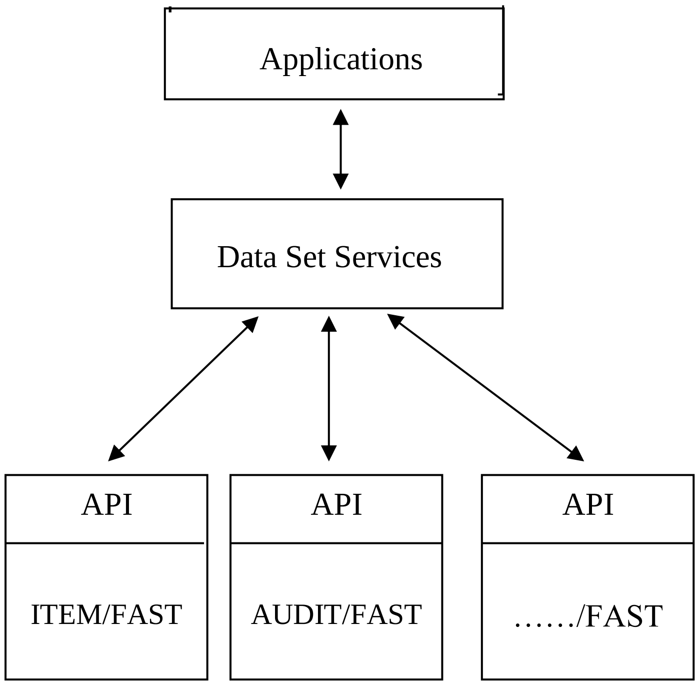


Figure 3.1

DSS to interact with FAST/TOOLS are called *DSS-clients.* PyTOOLS enables you to write DSS-clients in Python.

3.2 Data Sets: tables and properties

So what exactly is a data set?. Data sets are the way in which DSS represents FAST/TOOLS data. Related data is grouped into data sets. A data set consists of 2 components:

- The data itself
- The data set properties.

The data set data is presented to the pyTOOLS programmer as a database table from which records can be read. The data set properties hold important information about the data in the table like, data types, indexing, permissions to update or modify data and much more.

3.2.1 Data Set tables

A data set table is modelled as a two-dimensional table with fixed length rows. The rows in a table are called the *records* and the columns are called the *fields*.

3.2.2 Data Set properties

The other main component of the data set model are the data set properties. These properties describe the characteristics of a data set, like its name, the names of the fields and the size of the fields.

3.3 Data set indexes

Each data set has one or more indexes. Indexes are used to uniquely identify records in a data set. They also control the order in which records in a dataset are sorted. An index can consist of a single dataset field or combination of fields.

Whether a field can be used as an index is determent during the design of the data set. A data set can have many indexes but only one can be the *active index* at any given time. The pyTOOLS program can change the active index and thereby changing the order in which records are read from the dataset. Each data set must have at least one index.

One index in each data set is the *primary index*. This index is used by default when a data set is opened. An index can also be defined as a *unique index*. When a data set field is a unique index that field must have a different value in each record in the data set.

3.3.1 Composed indexes

In DSS an index can consist of more than one dataset field, such an index is called a *composed index*. All pyTOOLS DSSmodule function that take an index value as parameter will either accept a simple variable as parameter or a composed data type that contains values for all the fields that are part of the index. The exact use of composed indexes is explained further on in this manual, together with the functions that use them.

CHAPTER 4

Using pyTOOLS

"Increasingly, people seem to misinterpret complexity as sophistication, which is baffling. The incomprehensible should cause suspicion rather than admiration. Possibly this trend results form a mistaken belief that using a somewhat mysterious device confers an aura of power on the user"
-- Niklaus Wirth

This chapter is intended to be a conversational introduction to the pyTOOLS package. Some previous experience with FAST/TOOLS is presumed. You don't need to have any knowledge of Python.

4.1 The Python environment

To run a Python program you must have the Python interpreter installed on your system. One of the great features of the Python interpreter is that it can also be used interactive, like a UNIX or MSDOS command shell.

4.1.1 IDLE

The standard Python distribution includes IDLE a very basic IDE (integrated development environment) written in Python!. It includes and editor, a debugger, a class browser and an interactive command prompt. I found it to be quite adequate, it is based on the tcl/tk library and makes no attempt to hide its proud Unix is origin. IDLE can be started from the Python program group in the windows Taskbar.

4.1.2 The PyCharm IDE

Although the IDLE tool is great to get you started with Python there are much more advanced IDE's available. If you start developing your python code you should definitely take a look at PyCharm. www.jetbrains.com/pycharm . This is very complete IDE that offers everything you expect of a modern development environment. There is a community edition available for free!

4.1.3 Using Python interactively

Once you started you Python environment the interactive window appears.

```
Python 2.7.3 (default, Apr 10 2012, 23:31:26)[MSC v.1500 32 bit (Intel)] on
win32
Type "copyright", "credits" or "license()" for more information.
>>>
```

You can now enter commands after the command prompt ">>>". You can enter any Python command and the result will be printed:

```
>>> 2 + 3
5
>>>
```

You can also use variables. If the result of an expression is assigned to a variable it is not printed.

```
>>> x = 3 + 3
>>> x/3
2
>>>
```

In case of an error the interpreter will print the contents of the stack and an error message:

```
>>> x/0
Traceback (innermost last):
File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo>
>>>
```

You can even enter multi-line constructs interactively:

```
>>> # Fibonacci series:
>>> # the sum of two elements defines the next
>>> a, b = 0, 1
>>> while b < 10:
...     print b
...     a, b = b, a+b
...
1
1
2
3
5
8
>>>
```

This little for-loop shows us another feature of the Python language; compound statements are grouped according to their indentation. All statements in the while-loop must be indented one tab position further than the while-command. So, don’t forget to put a tab in front of the statements in the while-loop!. Although this might look unusual at first it will quickly become a second nature and result in tidy and readable code. Or like we Python programmers say: ‘*Live is better without braces!.’*

The rest of the chapter will focus on the use of the pyTOOLS DSS-module. If you want to learn more about Python syntax, which I am sure you want, there is an excellent tutorial in the standard Python documentation.

Python syntax is very readable and intuitive so you won’t have any problem understanding the examples that follow.

4.2 Introduction to pyTOOLS

For the examples in this intro to work you must have FAST/TOOLS running, Python 2.7 and the PyTOOLS extension module installed. Some examples return very long lists that can’t be shown completely. In that case dots . . . . . will indicate that the list is shortened.

Now let’s import the DSS-module so we can interact with FAST/TOOLS.

```
>>> import dss
>>>
```

The command 'import' tells the interpreter to load an external module. The interpreter will look for a dynamic library file with the name dss.pyd on a windows platform or dss.so on a UNIX platform. The dss.pyd file is just a normal windows dll file. The extension '.pyd' is only used to distinguish Python extension modules form all the other dll files on a windows system.

Now that the dss module has been imported we can connect to de FAST/TOOLS DSS-server and open a data set. To connect to DSS enter the command:

```
>>> conn = dss.connect()
>>>
```

The function connect() returns a connection handle that is used for any interaction with the FAST/TOOLS DSS-server. To get a list of all available datasets we can type:

```
>>> dss.getAvailableDatasets(conn)
['AUTH_ACTION_DF', 'AUTH_GROUP_DF', 'USER_DF', 'ALARM_ACK_DF', 'ALARM_AOI_DF',
'ALARM_ASA_DF', 'ALARM_DISPLAY_DF', 'ALARM_FO_DF', 'ALARM_FU_DF',
'ALARM_GROUP_DF, . . . ]
>>>
```

This example introduces us to an important data type: *the list*. Python makes extensive use of lists. Lists are written as comma-separated items between square brackets. A list can hold objects of any type, including other list. The list above only contains strings. Lists can be indexed, sliced, sorted, iterated over and much more. Like C arrays, lists are indexed by numbers. For more information on the Python lists please see the standard tutorial. To get a list of the fields in a particular data set enter the command:

```
>>> dss.getFieldNames(conn,'USER_DF')
['NAME', 'DESCRIPTION', 'DISPLAY_AUTH_LEVEL', 'PASSWORD', 'AUTH_GROUP', 'ASA',
'INITIAL_DISPLAY', 'PROCESS_AREAS']
>>>
>>>
```

This prints a list of all fields in the dataset USER_DF. Off course we can assign the result of the getFieldNames() function to a variable:

```
>>> fields = dss.getFieldNames(conn,'USER_DF')
>>> fields
['NAME', 'DESCRIPTION', 'DISPLAY_AUTH_LEVEL', 'PASSWORD', 'AUTH_GROUP', 'ASA',
'INITIAL_DISPLAY', 'PROCESS_AREAS']
>>>
```

And we can iterate over the resulting list object:

```
>>> for field in fields:
..     print field
NAME
DESCRIPTION
DISPLAY_AUTH_LEVEL
PASSWORD
AUTH_GROUP
ASA
INITIAL_DISPLAY
```

```
PROCESS_AREAS
>>>
```

Next we will open the data set USER_DF for reading. This is done by calling the function openDataset()

```
>>> dataset = dss.openDataset(conn, 'USER_DF',['NAME','DESCRIPTION'], 'r')
>>>
```

The openDataset function takes four arguments, the DSS connection handle, the name of the dataset, a list of field names, and a string indicating in which 'mode' the dataset will be opened. The ‘r’ indicates open for reading. Other options are ‘i’ for insert, ‘u’ for update and ‘d’ for delete, or any combination of these. The function returns a data set handle. We can now read records from the dataset using the readNext() function, the result of the read action depends off course on the users that are defined on your FAST/TOOLS system.

```
>>> dss.readNext(conn, dataset)
{'NAME': 'SYSMAN', 'DESCRIPTION': 'The system manager'}
>>>
```

This example assumes that you have at least one user defined in your FAST/TOOLS application. If there are no users the readNext function will raise an exception. This will look like this:

```
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in ?
    dss.readNext(conn, dataset)
error: ('DSS-E-EOS', 'Reached end of data set')
```

Later on in this manual the use of exceptions and how to handle them is explained in more detail.

The *readNext* function returns a dataset record in another much used Python compound data type: *the dictionary*. Dictionaries are sometimes found in other languages as "associative memories" or "associative arrays". Unlike lists, which are indexed by of numbers, dictionaries are indexed by keys. It is best to think of a dictionary as an unordered set of key:value pairs, with the requirement that the keys are unique (within one dictionary). So let’s read the next record from the dataset and see what we can do with a dictionary:

```
>>> record = dss.readNext(conn, dataset)
>>>
```

The dictionary 'record' has now the following contents:

```
{'DESCRIPTION': 'The operator',
 'NAME'       : 'OPERATOR'
```

The keys are formed by the field names and the values are the field values. You can reach the value fields by their key. Example:

```
>>> record['NAME']
'OPERATOR'
>>>
```

When you are ready using the data set you should close it to free system resources:

```
>>> dss.closeDataset(conn, dataset)
>>>
```

To close the connection to the DSS server type:

```
>>> dss.close(conn)
>>>
```

This has been a very short introduction into the possibilities of Python and the pyTOOLS-DSS extension module. There are many more-functions available, they are described in chapter 5. To get an overview of the available functions you can always use the help() function from the interactive command line:

```
>>> import dss
>>> help(dss)

NAME
    dss - This module provides access to the FAST/TOOLS Data Set Services
         (DSS).

FILE
    c:\python27\dlls\dss.pyd

FUNCTIONS
    activateFilter(...)
        activateFilter(conn_hdl, dataset_hdl) -- Returns None.

        This function activates the filter that was previously created with the
        createFilter function.
        conn_hdl is the DSS connection handle.
        dataset_hdl is the dataset handle.

    busReceive(...)
        busReceive(conn_hdl, [, 'SYNC_WAIT']) -- Returns a BUS/FAST message or
        one.

        Checks if there us a BUS/FAST message waiting if not returns None.
        conn_hdl is the DSS connection handle.
        Optional arg 'SYNC_WAIT', If present the function will not return
        until a message is received

    busReply(...)
        busReply(conn_hdl, message) -- Returns None.

        Reply to a BUS/FAST message.
        conn_hdl is the DSS connection handle.
        message is the reply message.

    busSend(...)
        busSend(conn_hdl, message) -- Returns None.

        Send a BUS/FAST message.
        conn_hdl is the DSS connection handle.
        message is the BUS/FAST message.
```

....

CHAPTER 5

The pyTOOLS-DSS interface

We can't stop people from complaining
but we can influence what they complain about.
-- Tim Peters

This chapter contains a formal description of all the functions in the pyTOOLS-DSS module.

5.1 General

In this chapter the pyTOOLS functions are grouped in sections according to their function. The groups are:

- Getting connected
- Get DSS properties
- Opening and closing data sets
- Reading records from a data sets
- Modifying data sets
- Indexes
- Filters
- Record locking
- Event notification
- Sending and receiving BUS/FAST messages
- Date and time functions
- Item history functions
- Miscellaneous functions

Each of these function groups is explained in a separate section in this chapter. Each section will have a short introduction to the purpose of the functions and an example. The examples can be entered interactively form the Python command prompt. Some examples return very long lists that can’t be shown completely. In that case dots . . . . . will indicate that the list is shortened.

5.2 Notational conventions

Throughout this chapter the following conventions are used to distinguish elements of text:

bold	Used for function names and literal portions of syntax that must appear exactly as shown.
italic	Used for filenames, variables and placeholders that represent the type of text to be entered by the programmer.

`monospace`	Used for sample program code and sample sessions
[ ]	Brackets surround optional parameters

5.3 Function index

Group	Function	Page
Getting connected	Connect to DSS	22
	Logging on	22
	Logging out	22
	Close connection	23
Get DSS properties	Get available data sets	24
	Get field names	24
	Get field properties	24
Opening and closing data sets	Open a data set	26
	Closing a data set	26
Reading records from a data sets	Read next record	28
	Read equal record	28
	Read a list of records	29
	Rewind data set	30
Modifying data sets	Insert record	31
	Insert a list of records	32
	Update record	33
	Update a list of records	34
	Delete record	35
Indexes	Get available indexes	36
	Set index	36
Filters	Create a filter	40
	Activate a filter	40
	Deactivate a filter	41
	Delete a filter	41
Record locking	Lock a record	42
	Unlock a record	42
Event notification	Request events on data set level	44
	Stop events on data set level	45
	Request events on record level	45
	Stop events on record level	45
	Reading events from queue	46
Using BUS/FAST	Set process name	49
	Sending a message	50
	Sending a message and wait for reply	51
	Receiving a message	52

5.4 Getting connected

5.4.1 General

Before pyTOOLS is able to use any of the other functions it must connect to DSS. A pyTOOLS program can connect only once to DSS. Trying to create more than one connection form the same program will result in an exception being raised. When pyTOOLS has successfully connected to DSS you get a ‘connection handle’. This handle needs to be preserved since it is needed by all of the other pyTOOLS functions.

DSS implements an authorisation scheme. This authorisation scheme is used to determine what kind of data can be retrieved and what kind of operations pyTOOLS can perform. Logging-on can be done at connection time or later in a separate action. When pyTOOLS does not perform a log-on, DSS will grant the most extensive permissions. When pyTOOLS performs a log-on, you have to pass a user name and a password. This user name will also be used by DSS when reporting events to AUDIT/FAST.

The pyTOOLS can log-on as many times as you wish. This might be useful to (temporary) grant a pyTOOLS program a more or less extensive permission level. PyTOOLS will not preserve context information in such a situation, i.e. successive log-on actions will overwrite previously specified information. When a log-on attempt fails, pyTOOLS client will remain it current authorisation.

5.4.2 Connect to DSS

Description:

Connects to DSS and returns a connection handle that is needed to communicate with the DSS server. If the optional *user* and *password* parameter are supplied a login to the DSS is done and access to data sets is limited to the permissions that were given to *user.*

Syntax:

conn_hnd = **dss.connect(**[*user*],[*password*]**)**

- conn_hnd
  Unique handle identifying the connection to DSS. The connection handle is a Python dictionary containing the DSS connection id and optionally the user name and password.
- user
  String containing the user name.
- password
  String containing the users password.

Returns:

A connection handle object.

5.4.3 Log-on to DSS

Description:

Logs-on to DSS. It enables pyTOOLS to (temporary) change the authorisation level and run under a different ‘user account’.

Syntax:

dss.logon (*conn_hnd,user,[password]***)**

- conn_hnd
  Unique handle identifying this connection to DSS.

- user
  String containing the user name.
- password
  String containing the users password.

Returns:

None.

5.4.4 Log-out of DSS

Description:

Logs-out of DSS. Authorisation level is restored to that from before logging in.

Syntax:

dss.logout(*conn_hnd***)**

- conn_hnd
  Unique handle identifying this connection to DSS.

Returns:

None.

5.4.5 Disconnect from DSS

Description:

Closes the connection to DSS. This enables DSS to free all resources used by pyTOOLS.

Syntax:

dss.close(*conn_hnd***)**

- conn_hnd
  Unique handle identifying this connection to DSS.

Returns:

None.

5.4.6 Examples

```
>>> import dss
>>> conn = dss.connect()
>>> dss.logon(conn, 'OPERATOR', 'OPERATOR')
>>> dss.logout(conn)
>>> dss.close(conn)
```

5.5 Get DSS properties

5.5.1 General

With these functions all kind of information at the data set level can be obtained.

5.5.2 Get available data sets

Description:

This function returns a list of all available data sets in DSS.

Syntax:

data_sets = **dss.getAvailableDatasets (***conn_hnd***)**

- data_sets
  List of strings containing the names of all available data sets.
- conn_hnd
  DSS connection handle.

Returns:

A list with the names of al available datasets in DSS.

5.5.3 Get data set fields names

Description:

Returns a list of the fields in a data set

Syntax:

fields = **dss.getFieldNames**(*conn_hnd, dataset*)

- fields
  List of strings containing the names of all available data sets.
- conn_hnd
  DSS connection handle.
- dataset
  Name of the data set, all fields in this data set will be returned by this function.

Returns:

A list with the names of al fields in a datasets in DSS

5.5.4 Get fields properties

Description:

Returns a dictionary of properties for a specific the field in a data set. The properties that are returned include

- Field name
- Field data type
- Field length in bytes

Syntax:

prop = **dss.getFieldProperties**(*conn_hnd, dataset,field*)

- prop
  Dictionary containing field properties.
- conn_hnd
  DSS connection handle.
- dataset
  Name of the data set, all fields in this data set will be returned by this function.
- field
  The fieldname.

Returns:

A dictionary with field properties.

5.5.5 Examples

```
>>> import dss
>>> conn = dss.connect()
>>> dss.getAvailableDatasets(conn)
['AUTH_ACTION_DF', 'AUTH_GROUP_DF', 'USER_DF', 'ALARM_ACK_DF',......]
>>>
>>> dss.getFieldNames(conn, 'USER_DF')
['NAME', 'DESCRIPTION', 'DISPLAY_AUTH_LEVEL','PASSWORD', ,......]
>>>
>>> prop = dss.getFieldProperties(conn,'USER_DF','NAME')
>>> print prop
>>> {'length': 32, 'name': 'NAME', 'type': 'CHAR'}
>>>
>>> prop['type']
>>> 'CHAR'
```

5.6 Opening and closing Data Sets

5.6.1 General

Prior to an operation on a data set, it needs to be opened. Opening a data set is like opening a data channel between DSS and your pyTOOLS program. Opening a certain data set can be done as many times as needed. After a data set has been successfully opened, the routine returns a *dataset handle* that is needed for subsequent operations on the data set.

When a dataset is no longer needed it should be closed. This enables DSS to free all resources occupied by that specific connection to the data set.

5.6.2 Open a data set

Description:

This function opens a dataset and returns a unique dataset handle.

Syntax:

dataset_hnd = **dss.openDataset**(*conn_hnd*, *dataset*, *fields*, *mode*)

- dataset_hnd
  Unique handle identifying this connection to the data set.
- conn_hnd
  The DSS connection handle.
- dataset
  Name of the data set to open.
- fields
  List of all field names. When opening a DSS data set you tell DSS in which specific fields of the data set you are interested. This is done to prevent unnecessary data communication.
- mode
  String indicating in which 'mode' the dataset will be opened. The mode can be read 'r', update 'u', delete 'd' , insert 'i' or any combination of those.

Returns:

A dataset handle.

5.6.3 Closing a data set

Description:

This function closes an open data set.

Syntax:

dss.closeDataset*(conn_hnd*, *dataset_hnd*)

- *conn_hnd*
  DSS connection handle.
- dataset_hnd
  Unique handle identifying this connection to the data set. The connection handle is a Python dictionary that contains, among other things, the data set connection id.

Returns:

None.

5.6.4 Examples

```
>>> import dss
>>> conn = dss.connect()
>>> # Open dataset USER_DF for read and update. We are interested in the fields
... # 'NAME' and 'PASSWORD' and we want to be able to read records and update
... # their value.
>>> data_set = dss.openDataset(conn, 'USER_DF', ['NAME', 'PASSWORD'], 'ru')
>>> # Close the dataset
>>> dss.closeDataset(conn, data_set)
>>>
```

5.7 Reading records from a data set

5.7.1 General

After a data set has been opened we can user the pyTOOLS read functions to retrieve data from it. There are three read function; *readNext*- which will read sequentially thru the data set, *readEqual*- which can be used to read a specific record from the data set and *readRecords* - which reads all records from a data set. A related function is the *rewindDataset* function. This function will set the 'current record' pointer to the first record in the data set. This means that the next time you call the readNext function it will start at the beginning of the data set.

5.7.2 Read next record

Description:

This routine returns the 'next' record in the data set. The order in which records are read is determent by the active index and possibly a filter. When the data set is opened the data set cursor is placed at the first record in the set. After each readNext action the cursor is moved to the next record. The function rewindDataset can be used to place the cursor on the first record of the dataset again. The readNext function returns a data set record as a Python dictionary where the keys are the records field names and the values are the record values.

Syntax:

record = **readNext(***conn_hnd, dataset_handle***)**

- *record*
  data set record.
- *conn_hnd*
  DSS connection handle.
- dataset_hnd
  Unique handle identifying this connection to the data set .

Returns:

A dictionary containing a dataset record.

Read next example:

```
>>> import dss
>>> conn = dss.connect()
>>> # Open dataset USER_DF for read
>>> data_set = dss.openDataset(conn, 'USER_DF',['NAME', 'PASSWORD'], 'r')
>>> # read the first record from data set.
>>> record = dss.readNext(conn,data_set)
>>> print record
{'PASSWORD': '*********', 'NAME': 'OPERATOR'}
>>>
```

5.7.3 Read equal record

Description:

This function is used to read a specific record from a data set. Data set records are identified by their index field. The readEqual function will search for a record in the data set where the index field is equal to the *field_value* argument. After each readEqual action the data set cursor is moved to the record that follows the one that was read.

This operation is performed using the index currently set (See also: paragraph 5.9.3 Set indexes). If the read operation is performed on an index that is not unique, the operation will return the first occurrence of the record satisfying the index value. No filter criteria apply with this type of data retrieval operation

Syntax:

record = **readEqual(***conn_hnd*, *dataset_handle*, *index_value***)**

- *record*
  Data set record.
- *conn_hnd*
  DSS connection handle.
- *dataset_hnd*
  Unique handle identifying this connection to the data set.
- *index_value*
  Index field value of the record you want to retrieve. In cause of a composed index this parameter can be a dictionary where the index fields names are the keys and the field values the value fields

Returns:

A dictionary containing a dataset record.

Read equal examples:

```
>>> import dss
>>> conn = dss.connect()
>>> # Open dataset USER_DF for read
>>> data_set = dss.openDataset(conn, 'USER_DF', ['NAME', 'PASSWORD'], 'r')
>>> # Read the record ‘SYSMAN’ from the data set.
>>> record = dss.readEqual(conn,data_set, 'SYSMAN')
>>> print record
{'PASSWORD': '*********', 'NAME': 'SYSMAN', 'DESCRIPTION': 'The manager'}
>>>
>>> # Example using composed index, first create an index dictionary
>>> index = { 'NAME' : 'SYSMAN'}
>>> record = dss.readEqual(conn,data_set, index)
>>> print record
{'PASSWORD': '*********', 'NAME': 'SYSMAN', 'DESCRIPTION': 'The manager'}
```

5.7.4 Read all record from a data set

Description:

This function is used to read all record from a data set at once. This can be more efficient than reading every record separately. This function returns a list of data set record, it is especially useful in combination with the function **updateRecords.**

Syntax:

records = **readRecords(***conn_hnd*, *dataset_handle***)**

- *records*
  List of data set records.
- *conn_hnd*
  DSS connection handle.
- *dataset_hnd*
  Unique handle identifying this connection to the data set.

Returns:

A list of dictionaries.

Read all records example:

```
>>> import dss
>>> conn = dss.connect()
>>> # Open dataset USER_DF for read
>>> data_set = dss.openDataset(conn, 'USER_DF',['NAME', 'PASSWORD'], 'r')
>>> records = dss.readRecords(conn, data_set)
>>> for record in records:
...    print record['NAME']
...
OPERATOR
SYSMAN
>>>
```

5.7.5 Rewind data set

Description:

This function places the data set cursor on the first record of a data set. This can be useful in conjunction with the readNext function. This means the next time you use the readNext function it will read the first record from the data set. Which record is the first record is determent by the active index.

Syntax:

dss.rewindDataset(*conn_hnd, dataset_handle***)**

- *conn_hnd*
  DSS connection handle.
- *dataset_hnd*
  Unique handle identifying this connection to the data set.

Returns:

None.

Rewind data set example:

```
>>> import dss
>>> conn = dss.connect()
>>> # Open dataset USER_DF for read
>>> data_set = dss.openDataset(conn, 'USER_DF',['NAME', 'PASSWORD'], 'r')
>>> dss.readNext(conn,data_set)
{'PASSWORD': '*********', 'NAME': 'OPERATOR'}
>>> dss.rewindDataset(conn, data_set)
>>> dss.readNext(conn,data_set)
{'PASSWORD': '*********', 'NAME': 'OPERATOR'}
```

5.8 Modifying data sets

5.8.1 General

The pyTOOLS DSS-module can be used to modify DSS data set records in the following way:

- Insert a new record in a data set
- Insert a list of records in a data set
- Update one or more fields in a data set record.
- Delete a data set record

5.8.2 Insert a new record in data set

Description:

Inserts a record in a data set. Only the fields that pyTOOLS has declared an interest in during the opening of the data set (**dss.openDataset**) can be given a value. All other fields are set to a default values.

Syntax:

dss.insertRecord(*conn_hnd, dataset_handle, record)*

- *conn_hnd*
  DSS connection handle.
- *dataset_hnd*
  Unique handle identifying this connection to the data set.
- *record*
  A dictionary that contains fieldname:value pairs, it must contain the current index field to uniquely identify the record.

Returns:

None.

Insert record example

```
>>> import dss
>>> conn = dss.connect()
>>> # Open the data set USER_DF for ‘read’ and ‘insert’
>>> data_set = dss.openDataset(conn, 'USER_DF',['NAME','AUTH_GROUP'], 'ri')
>>>
>>> # Create an empty dictionary
>>> new_record = {}
>>>
>>> # Fill record with data
>>> new_record['NAME']= 'SUPER_USER'
>>> new_record['AUTH_GROUP'] = 'VERY HIGH'
>>>
>>> # Insert new record in data set
>>> dss.insertRecord(conn, data_set, new_record)
>>>
>>> # Check record has been inserted
>>> dss.readEqual(conn,data_set, 'SUPER_USER')
{'AUTH_GROUP': 'VERY HIGH', 'DESCRIPTION': '', 'NAME': 'SUPER_USER',
'PASSWORD': '******************************'}
```

5.8.3 Insert a list of record in data set

Description:

This function inserts a list of record in a data set. Only the fields that pyTOOLS as declared an interest in during the opening of the dataset (**dss.openDataset**) can be given a value. All other values are set to a default values.

Syntax:

dss.insertRecords(*conn_hnd, dataset_handle, records)*

- *conn_hnd*
  DSS connection handle.
- *dataset_hnd*
  Unique handle identifying this connection to the data set.
- *records*
  A list of dictionary that contains fieldname:value pairs, it must contain the current index field to uniquely identify the record.

Returns:

None.

Insert records example:

```
>>> import copy
>>> import dss
>>> conn = dss.connect()
>>> # Open the data set USER_DF for read and insert
>>> data_set = dss.openDataset(conn, 'USER_DF',['NAME','AUTH_GROUP'], 'ri')
>>>
>>> # Create an empty list and a dictionary
>>> record_list = []
>>> new_record = {}
>>>
>>> # Fill record with data and place it in the list
>>> for i in range(10):
. . .    new_record['NAME']= 'USER_' + str(i)
. . .    new_record['AUTH_GROUP'] = 'VERY HIGH'
. . .    record_list.append(copy.copy(new_record))
>>>
>>> # Insert new record in data set
>>> dss.insertRecords(conn, data_set, record_list)
>>>
>>> # Check if records have been inserted
>>> for i in range(10):
. . .    record = dss.readEqual(conn, data_set, 'USER_' + str(i))
. . .    print record['NAME']
USER_0
USER_1
USER_2
USER_3
USER_4
USER_5
USER_6
USER_7
USER_8
USER_9
```

Note: The previous example makes use of the Python copy module to append a copy of the dictionary ‘new_record’ to the list ‘record_list’. The reason for this is that in Python every variable is a reference to an object. When we append a record to a list all we really do is append a reference to that record to the list. If we didn’t insert a copy of the record we would insert 10 references to the same record! The copy function creates a new object. Exanple:

```
>>> list_1 = ['a']
>>> list_2 = list_1
>>> list_2
['a']
>>> list_1[0] = 'b'
>>> list_2
['b']
>>> import copy
>>> list_2 = copy.copy(list_1)
>>> list_2
['b']
>>> list_1[0] = 'c'
>>> list_2
['b']
>>>
```

5.8.4 Update record in a data set

Description:

Updates record field values.

Syntax:

dss.updateRecord(*conn_hnd, dataset_handle, record)***)**

- *conn_hnd*
  DSS connection handle.
- *dataset_hnd*
  Unique handle identifying this connection to the data set.
- *record*
  Python dictionary that contains fieldname:value pairs, it must contain the current index field to uniquely identify the record.

Returns:

None.

Update record example:

```
>>> import dss
>>> conn = dss.connect()
>>>
>>> # Open the 'USER_DF' data set and read the record 'SYSMAN'
>>> data_set = dss.openDataset(conn, 'USER_DF',['NAME','DESCRIPTION'], 'ru')
>>> record = dss.readEqual(conn, data_set, 'SYSMAN')
>>>
```

The read equal function return a dictionary containing fiedname:value pairs, the contents of record is now

```
'DESCRIPTION': 'The system manager',
'NAME': 'SYSMAN',
```

To update the value of a field just change its value and update the dataset:

```
>>> record['DESCRIPTION'] = 'New description'
>>> dss.updateRecord(conn, data_set, record)
```

The dataset has now been updated

```
>>> dss.readEqual(conn, data_set, 'SYSMAN')
{'NAME': 'SYSMAN', 'DESCRIPTION': 'New description'}
>>>
```

Of cause we can create a field value dictionary without using the readEqual function:

```
>>> # Create a empty dictionary and fill it with the fields you want to update
>>> field_dict = {}
>>> field_dict['NAME'] = 'SYSMAN'
>>> field_dict['DESCRIPTION'] = 'New description'
>>> dss.updateRecord(conn, data_set, field_dict)
```

5.8.5 Update a list of records

Description:

This function takes a list of data set records and updates the them in the data set

Syntax:

dss.updateRecords(*conn_hnd, dataset_handle, records)*

- *conn_hnd*
  DSS connection handle.
- *dataset_hnd*
  Unique handle identifying this connection to the data set.
- *records*
  A list of dictionary that contain fieldname:value pairs to update, it must contain the current index field to uniquely identify the record.

Returns:

None.

Update records example

```
>>> import dss
>>> conn = dss.connect()
>>>
>>> # Open the USER_DF data set and read the all records
>>> data_set = dss.openDataset(conn, 'USER_DF',['NAME','DESCRIPTION'], 'ru')
>>> records = dss.readRecords(conn, data_set)
>>>
>>> # Read update DESCRIPTION field and update data set
>>> for record in records:
>>>      record['DESCRIPTION'] = 'New description'
>>>
>>> dss.updateRecords(conn, data_set, records)
```

5.8.6 Delete data set record

Description:

Deletes a record from a data set. The record is identified by the current index.

Syntax:

dss.deleteRecord(*conn_hnd, dataset_handle, index***)**

- *conn_hnd*
  DSS connection handle.
- *dataset_hnd*
  Unique handle identifying this connection to the data set.
- *index*
  Index value of the record you want to delete. In cause of a composed index this parameter can be a dictionary where the index fields names are the keys and the field values the value fields

Returns:

None.

Delete record example:

```
>>> import dss
>>> conn = dss.connect()
>>>
>>> data_set = dss.openDataset(conn, 'USER_DF',['NAME','PASSWORD'], 'rd')
>>> dss.deleteRecord(conn, data_set, 'SUPER_USER')
```

The record 'SUPER_USER' is now removed from USER_DF:

```
>>> dss.readEqual(conn,data_set, 'SUPER_USER')
Traceback (innermost last):
File "<interactive input>", line 1, in ?
error: DSS-E-EOS, Reached end of data set
```

```
>>> # Example using composed index, first create an index dictionary
>>> index = { 'NAME' : 'USER_USER'}
>>> dss.deleteRecord(conn, data_set, index)
```

5.9 Indexes

5.9.1 General

Each data set has one or more indexes. Indexes are (in most cases) data set fields like any other data set field but they have some special functions. Index-fields can be used to identify records in a data set and they can be used to control the order in which records in a data set are sorted.

Whether a field is an index-field or not is determent during the design of the data set. A data set can have many indexes but only can be the *active index* at any given time.

Before doing a read operation, pyTOOLS can set the index to use during the read operation. The index determines in which order records will be read from the data set. Each data set has at least one unique index. PyTOOLS refers to an index by the index name. When pyTOOLS has not explicitly set an index DSS will use the primary index.

The pyTOOLS programmer can change the active index (using dss.setIndex) and thereby changing the order in which records are read from the data set.

5.9.2 Get available indexes

Description:

This function returns a list containing all possible indexes for a data set.

Syntax:

index_list = **dss.getIndex(***conn_hnd, dataset_name***)**

- *index_list*
  List containing all possible indexes.
- *conn_hnd*
  DSS connection handle.
- *dataset_name*
  String containing the dataset name.

Returns:

A list off index names.

5.9.3 Set indexes

Description:

This function sets the index for a dataset.

Syntax:

dss.setIndex(*conn_hnd, dataset_hnd, index_name* **)**

- *conn_hnd*
  DSS connection handle.
- *dataset_hnd*
  Unique handle identifying this connection to the data set.
- *index_field*
  Make *index_name* the active index.

Returns:

None.

5.9.4 Indexes example

```
>>> import dss
>>> conn = dss.connect()
>>>
>>> #Get a list of the available indexes for the dataset 'STATUS_DF'
>>> dss.getIndex(conn, 'STATUS_DF')
['NAME', 'STATUS_NUMBER']
>>>
>>> # Open STATUS_DF DATASET
>>> ds = dss.openDataset(conn, 'STATUS_DF',['NAME','STATUS_NUMBER'],'r')
>>>
>>> # The primary index is ‘NAME’ so this will be the active index after
>>> # opening the dataset. Read the first 5 records they are sorted on NAME
>>> for i in range(5):
...   dss.readNext(conn,ds)
...
{'STATUS_NUMBER': 101L, 'NAME': 'ALARM'}
{'STATUS_NUMBER': 105L, 'NAME': 'ALARMLOCAL'}
{'STATUS_NUMBER': 134L, 'NAME': 'AL_AK_NRS'}
{'STATUS_NUMBER': 135L, 'NAME': 'AL_AK_RS'}
{'STATUS_NUMBER': 132L, 'NAME': 'AL_NAK_NRS'}
>>>
>>> # Now change index to 'STATUS_NUMBER' and records will be sorted by
>>> # STATUS_MUMBER
>>> dss.setIndex(conn, ds, 'STATUS_NUMBER')
>>>
>>> for i in range(5):
...   dss.readNext(conn,ds)
...
{'STATUS_NUMBER': 0L, 'NAME': 'ZERO'}
{'STATUS_NUMBER': 1L, 'NAME': 'CALCULATE'}
{'STATUS_NUMBER': 2L, 'NAME': 'USE OLD'}
{'STATUS_NUMBER': 3L, 'NAME': 'BLOCKED'}
{'STATUS_NUMBER': 4L, 'NAME': 'BOOLEAN 0'}
>>>
>>>
```

5.10 Filters

5.10.1 General

Filters are used to avoid unnecessary data traffic between pyTOOLS and the DSS server. A filter gives you the possibility to tell the DSS server in which records of a dataset you are interested. Once you have placed a filter on a dataset the dss.readNext function will only return those records that match the filter criteria.

A filter is defined in a so-called filter expression. A filter expression consists of a string that must be matched and possibly one or more wildcard characters. During a read action the index field of the dataset to read must match the filter expression. The available wildcard characters are:

?	A question mark in a filter expression means that any character can occupy that position.
*	An asterisk in a filter expression means that any character can occupy that position and any of the remaining positions in that filter.

Using a filter involves four dss-functions:

Create filter:	Before you can use a filter it must be created. Creating a filter means the filter expression syntax is checked and the filter is stored with associated with a dataset.
Activate filter:	Once you have created a filter for at data set you can turn it on using the activateFilter function.
Deactivate filter:	You can turn it of with deactivateFilter functions.
Delete filter	If you no longer need a filter you should delete it to free system resources.

Only one filter can be created at a time for a data set. You must delete any existing filter before you can create a new one.

5.10.2 Filter expression syntax

This section gives a formal description of the syntax for the DSS filter expressions. The following symbols and conventions are used for the syntax description:

• **Bold** printed elements are part of the language itself.
• [ ]
Means: enclosed item is optional.
• { }..
Means: enclosed item(s) may be repeated one or more times.
• < >:
Means: Enclosed item is defined as ....
• < >
Means: use definition of enclosed item.

An filter expression has one out of two possible layouts:

- <wildcards string constant>
- <logical expression>

whereby:

<wildcards string constant>:

"[{<wc char>}..]"

<wc char>:

- All printable characters, including ’*’ and ’?’.
- The character ’*’ is meant to indicate: zero or more characters.
- The character ’?’ is meant to indicate: exactly one character.

The <wildcards string constant> is a short notation for the
<logical expression>: **NAME LIKE** <wildcards string constant>

<logical expression>:

<comparison> [{<logical operant> <comparison>}..]

<comparison>:

<field name> <comparison operator> <constant value>

<field name>:

Name of a field.

<comparison operator>:

One of:

- **=**
- **!=**
- **<**
- **<=**
- **>**
- **>=**
- **LIKE**

<constant value>:

<numeric constant> | <string constant>

<numeric constant>:

For example: **12.4**

<string constant>:

For example: **"INS.UNI.VALVE2"**

<logical operant>:

One of:

- **OR**
- **AND**

Examples of logical expressions are:

NAME LIKE "INS.*" (is same as: **"INS.*"**)

(ITEM_TYPE = "Output" OR ITEM_TYPE = "Input") AND HIGH_LIMIT > 100.2

Note that string constants are case sensitive and that field names and operators are case insensitive.
Also note that DSS requires you to put spaces between operators and expressions.

5.10.3 Create a filter

Description:

Creates a filter for a data set. If your filter expression consists of a single string constant (with or without a wildcard) than the filter expression is applied to the current index field of the dataset. For example, if your dataset is ITEM_DF then the filter is applied to the NAME field because that's the default index field, so the expression:

    '"*.UNIT_2.*"'

is true for all records where unit is UNIT_2. You have probably noticed the use of both single and double quotes in this expression. This is necessary because "*.UNIT_2.*" must be seen as a string inside a string. The filter expression itself is a string and *.UNIT_2.* is also a string inside that string. This is better explained for a more complex expression:

    '(ITEM_TYPE = "Output" OR ITEM_TYPE = "Input") AND HIGH_LIMIT > 100.2'

Here single quotes are used to contain the complete filter expression and the double quotes are used to contain the string constants. If we had used the same type of quotes in both cases the DSS filter software would be unable to determine it the presents of a quote indicated the end of a filter expression or the start of a string within that same expression. The use of a string inside a string is a problem for every programming language. Python has solved this problem very elegant by allowing both single and double quotes to delimit strings.

Syntax:

dss. createFilter(*conn_hnd, dataset_hnd, filter_expression*)

-
  DSS connection handle.
- *dataset_hnd*
  Unique handle identifying this connection to the data set.
- *filter_expression*
  String containing the filter expression.

Returns:

None.

5.10.4 Activate filter

Description:

This function activates the filter that was previously created with the createFilter function. After this function is called only records that pass the filter criteria are read.

Syntax:

dss. activateFilter(*conn_hnd, dataset_hnd*)

- *conn_hnd*
  DSS connection handle.
- *dataset_hnd*
  Unique handle identifying this connection to the data set.

Returns:

None.

5.10.5 Deactivate filter

Description:

This function deactivates the filter.

Syntax:

dss. deactivateFilter(*conn_hnd*, *dataset_hnd*)

- *conn_hnd*
  DSS connection handle.
- *dataset_hnd*
  Unique handle identifying this connection to the data set.

Returns:

None.

5.10.6 Delete filter

Description:

Deletes a filter. You cannot remove an active filter.

Syntax:

dss. deleteFilter(*conn_hnd*, *dataset_hnd*)

- *conn_hnd*
  DSS connection handle.
- *dataset_hnd*
  Unique handle identifying this connection to the data set.

Returns:

None.

5.10.7 Filtering example

```
>>> import dss
>>> conn = dss.connect()
>>> fields = dss.getFieldNames(conn, 'ITEM_DF')
>>> ds = dss.openDataset(conn, 'ITEM_DF',fields, 'ru')
>>>
>>># We only want to read items from the unit 'UNIT_2'
>>> dss.createFilter(conn, ds,  '”*.UNIT_2.*”')
>>> dss.activateFilter(conn, ds)
>>> dss.readNext(conn,ds)
{'DEADBAND': 0.0, 'LOW_LOW_LIMIT': 0.0, 'NAME': 'INSTAL.UNIT_2.TAG1',........
>>>
>>> dss.deactivateFilter(conn,ds)
>>> dss.deleteFilter(conn,ds)
```

5.11 Record locking

5.11.1 General

By locking a record in a data set you prevent any other FAST/TOOLS process from modifying or deleting it. If the record is already locked by another FAST/TOOLS process an exception is raised.

5.11.2 Locking record

Description:

This function locks one or more records.

Syntax:

dss.lockRecords(*conn_hnd, dataset_hnd, records*)

- *conn_hnd*
  DSS connection handle.
- *dataset_hnd*
  Unique handle identifying this connection to the data set.
- *records*
  A list containing the primary indexes to records you want to lock.

Returns:

None.

5.11.3 Unlocking record

Description:

This function unlocks one or more records.

Syntax:

dss.unlockRecords(*conn_hnd, dataset_hnd, records*)

- *conn_hnd*
  DSS connection handle.
- *dataset_hnd*
  Unique handle identifying this connection to the data set.
- *records*
  A list containing the primary indexes to records you want to unlock.

Returns:

None.

5.11.4 Record locking example

```
>>> import dss
>>> conn =dss.connect()
>>> fields = dss.getFieldNames(conn, 'ITEM_DF')
>>> ds = dss.openDataset(conn, 'ITEM_DF',fields, 'ru')
>>> dss.lockRecords(conn, ds, ['INSTAL.UNIT_2.TAG1', 'INSTAL.UNIT_2.TAG2'])
>>> # Two records of the ITEM_DF dataset are now locked.
>>> # Try changing them in USER/FAST !
>>>
>>> dss.unlockRecords(conn, ds, ['INSTAL.UNIT_2.TAG1', 'INSTAL.UNIT_2.TAG2'])
>>>
>>># Two records of the ITEM_DF dataset are now unlocked.
```

5.12 Event notification

5.12.1 General

The pyTOOLS DSS-module implements an event notification scheme. After a program has declared an interest in changes to the contents of a data set it can poll the DSS server for events.

There are two levels of event criteria on which a program can request event notification:

- Data Set level
  Operations on the data set are globally monitored by DSS. As soon as DSS detects a change in the contents of the data set, the pyTOOLS program is notified.
- Data Set record level
  The DSS client can request for monitoring of changes on specific data set records. As soon as DSS detects a change of such a specific data set record, the pyTOOLS program is notified.

The overview below shows the possible events for each of the levels mentioned above.

Data Set	- record inserted
	- record deleted
	- record updated
Record	- record deleted
	- record updated

Different event criteria can be specified in parallel. For similar situations however, an event will only be sent once. Example: Suppose a pyTOOLS program requested notification of changes to a data set by specifying the criteria ‘record updated’ at both data set level and record level. Now suppose the contents of the record in question changes. In that situation only one event will be sent.

Event notification can only requested once for the same record. If you want to change the event notification criteria for a data set you have to stop the existing event notification and apply a new one.

The following section describes the pyTOOLS notification function. These will be a single example at the end of this section.

5.12.2 Request event notification on data set level

Description:

This function requests notification of changes to the contents of a dataset.

Syntax:

dss.datasetEventNotify*(conn_hnd, datase_handle, mode)*

- conn_hnd
  The DSS connection handle.
- dataset_handle
  Data set handle of the dataset
- mode
  String indicating in which dataset events you are interested. The mode can be update 'u', delete 'd' , insert 'i' or any combination of those.

Returns:

None.

5.12.3 Stopping event notification on data set level

Description

This function stops notification of changes to the contents of a dataset.

Syntax:

dss.datasetEventStop*(conn_hnd, datase_handle)*

- conn_hnd
  The DSS connection handle.
- dataset_handle
  Data set handle of the dataset

Returns:

None.

5.12.4 Request event notification on data set record level

Description:

This function requests notification of changes to a specific record in a dataset.

Syntax:

dss.recordEventNotify*(conn_hnd, datase_handle, prime_index, mode)*

- conn_hnd
  The DSS connection handle.
- dataset_handle
  Data set handle of the dataset
- *prime_index*
  Primary index field value(s) of the data set record for which event notification is required. In cause of a composed index this parameter can be a dictionary where the index fields names are the keys and the field values the value fields
- mode
  String indicating in which dataset record events you are interested. The mode can be update 'u', delete 'd' or both.

Returns:

None.

5.12.5 Stopping event notification on data set record level

Description:

This function removes requests notification for a specific record in a dataset.

Syntax:

dss.recordEventStop*(conn_hnd, datase_handle, prime_index)*

- conn_hnd
  The DSS connection handle.
- dataset_handle
  Data set handle of the dataset
- *prime_index*
  Primary index field value(s) of the data set record for which event notification is required. . In cause

of a composed index this parameter can be a dictionary where the index fields names are the keys and the field values the value fields

Returns:

None.

5.12.6 Reading events form the event queue

Description:

This function reads events form the DSS event queue. When a program has requested event notification DSS will write a message in the notification queue whenever an event occurs. It is the responsibility op de pyTOOLS program to check regularly for new events. It the number of messages in the queue exceeds a maximum DSS will clear the queue. The size of the event queue is defined in the DSS set-up file. Once you have read an event messages it is removed from the buffer. There is only one event queue per DSS-client so all events are stored in the same queue. The ‘dss.eventRead’ function returns a list of event messages from in the queue. If the queue is empty it will return an empty list.

Syntax:

event_list = ***dss.eventRead****(conn_hnd)*

- event_list
  List of dictionaries, each dictionary contains an event message.
- conn_hnd
  The DSS connection handle.

Returns:

A list of event records.

Event message layout:

The event message layout requires some explanation. Below we see a typical event message. The message is stored in a dictionary.

```
[{'ds_name': 'USER_DF',
  'event': 'u',
  'id': 6L,
  'record': {'ASA': '',
             'AUTH_GROUP': 'ALLES',
             'DESCRIPTION': '',
             'DISPLAY_AUTH_LEVEL': 40L,
             'INITIAL_DISPLAY': '',
             'NAME': 'JOHN',
             'PASSWORD': '********************************',
             'PROCESS_AREAS': ''},
  'set_hnd': 2717908993L}]
```

It contains the following fields:

- ds_name
  The dataset name for the dataset that was modified
- event
  The type of event that took place on the dataset. This can be update 'u', delete 'd' and in case of a notification on dataset level also insert 'i' , to indicate the record was inserted.

- id
  This is a unique sequence number given by pyTOOLS . The number six indicates this as the sixth number that was read from the event queue.
- record
  This is a dictionary holding the current value (after the event) of the record. It is the same type of record you would get from a dss.readNext( ) or dss.readEqual( ) function call.
- set_hnd
  This is the handle of the dataset that from which the event notification request was issued.

5.12.7 Event notification example

```
>>>
>>> import dss
>>> import pprint
>>> p = pprint.pprint
>>>
>>> conn = dss.connect()
>>> fields = dss.getFieldNames(conn, 'STATUS_DF')
>>> ds = dss.openDataset(conn, 'STATUS_DF', fields, 'riud')
>>>
>>> # Request notification on all events on the STATUS_DF dataset
>>> dss.datasetEventNotify(conn, ds, 'idu')
>>>
>>> # read one record from the dataset
>>> rec = dss.readNext(conn, ds)
>>> p(rec)
{'ALARM_PRIO': 0L,
 'ALARM_STATE': 'Alarm 1',
 'DESCRIPTION': 'alarms, ackd, not reset',
 'NAME': 'AL_AK_NRS',
 'STATUS_NUMBER': 134L,
 'USER_INFO': ''}
>>>
>>> # Change the USER_INFO field in the record and write it back
>>> # this will cause DSS to update the record and put a
>>> # message in the event queue.
>>> rec['USER_INFO'] = 'ALARM STATE ONE'
>>> dss.updateRecord(conn, ds, rec)
>>>
>>> # Check the contents of the event queue
>>> event_list = dss.eventRead(conn)
>>> p(event_list)
[{'ds_name': 'STATUS_DF',
  'event': 'u',
  'id': 0L,
  'record': {'ALARM_PRIO': 0L,
             'ALARM_STATE': 'Alarm 1',
             'DESCRIPTION': 'alarms, ackd, not reset',
             'NAME': 'AL_AK_NRS',
             'STATUS_NUMBER': 134L,
             'USER_INFO': 'ALARM STATE ONE'},
  'set_hnd': 2717908993L}]
>>>
>>> # The event queue is now empty
>>> event_list = dss.eventRead(conn)
>>> p(event_list)
[]
>>> len(event_list)
0
>>> # Once we have stopped event notification no further updates
>>> # to the dataset will be placed in the event queue
>>> dss.datasetEventStop(conn,ds)
>>> rec['USER_INFO'] = 'ALARM STATE TEST'
>>> dss.updateRecord(conn, ds, rec)
>>> event_list = dss.eventRead(conn)
>>> p(event_list)
[]
>>>
>>>
```

The next example shows a script that request an update notification for two items and then checks the event queue every 10 second. It can be stopped by pressing <ctrl> c.

```
import dss
import time
conn = dss.connect()
fields = dss.getFieldNames(conn, 'ITEM_VAL')
ds = dss.openDataset(conn, 'ITEM_VAL', fields, 'r')

# Request notification on value updates for two items
dss.recordEventNotify(conn, ds, 'INSTAL.UNIT.TAG1','u')
dss.recordEventNotify(conn, ds, 'INSTAL.UNIT.TAG2','u')

# Check event queue every 10 seconds, catch <ctrl> C
#  event from the keyboard to quit.
try:
    while 1:
        time.sleep(10)
        event_list = dss.eventRead(conn)
        for event in event_list:
            record = event['record']
            print record['NAME'], ' : ', str(record['ITEM_VALUE'])
except KeyboardInterrupt, e:
    pass

# Stop notification and close data set and DSS connection.
dss.recordEventStop(conn,ds,'INSTAL.UNIT.TAG1')
dss.recordEventStop(conn,ds,'INSTAL.UNIT.TAG2')
dss.closeDataset(conn,ds)
dss.close(conn)
```

5.13 Using BUS/FAST

5.13.1 General

FAST/TOOLS comes with its own protocol for inter-process communication called BUS/FAST. It is used for both data exchange and remote procedure calls among processes. Although not strictly a part of DSS, it forms the fundament of FAST/TOOLS and no DSS extension would be complete without support for BUS/FAST. So pyTOOLS has built-in support for BUS/FAST.

BUS/FAST is implemented in C and messages among processes are passed as C-structs. BUS/FAST messages in pyTOOLS can only contain character strings.

5.13.2 Setting BUS/FAST process name

Description:

Each FAST/TOOLS process, including any pyTOOLS script you write, must first make itself known to BUS/FAST before it can communicate with other FAST/TOOLS processes. In case of a pyTOOLS script you don't need to worry about this because it is automatically done for you when you connect to DSS. Each process that makes use of BUS/FAST gets a unique BUS/FAST-process name. This name can subsequently be used to address the process when sending and receiving messages. If you don't specify a specific name BUS/FAST will create a name for you. It will look something like 'PYDSS_12323', where the number part is unique. This name will be different each time your script connects to BUS/FAST.

A BUS/FAST generated process name will work fine for any pyTOOLS script that doesn't send messages to other processes. If however you want to send and receive messages from and to other scripts or FAST/TOOLS processes you should always use the same name when connecting to BUS/FAST. This makes it possible for other processes to address you.

You can tell pyTOOLS to use a specific name by calling the function 'busSetName' *before* you connect to DSS. You have to call this function each time you attach to BUS/FAST. If you detach from BUS/FAST the process name is cleared.

Syntax:

***dss.*busSetName** *(process_name)*

- *process_name*
  String containing the BUS/FAST process under which you want your process to attach to BUS/FAST.

Returns:

None

Set BUS/FAST process name example:

```
>>>
>>> # The following lines of code demonstrate
>>> # how to connect to BUS/FAST using a specific name.
>>> # After this other processes know your process as 'MY_PROC'.
>>>
>>> import dss
>>> dss.busSetName('MY_PROC')
>>> con = dss.connect()
>>>
```

5.13.3 Sending a BUS/FAST message

Description:

Once your processes has attached to BUS/FAST you can let it send messages to other FAST/TOOLS processes. Instead of C-structs, pyTOOLS uses Python dictionaries to send messages. Each message contains an address part and a messages part. The address contains the name and the node number of the receiving process. The message part consists of a message code and a message string. This sounds more complicated than it actually is. Just look at an example message:

```
my_message = {'process' : 'receiver',
              'node'    : 162,
              'code'    : 100,
              'message' : 'HELLO RECEIVER !' }
```

In the lines above a Python dictionary is created. All keys are strings. The fields have the following contents:

- process:
  Holds the name of the receiving process. This message is send to a FAST/TOOLS process called ‘receiver’. This field must by a string data type.

- node:
  In case of a multi-node FAST/TOOLS system this field contains node number of the receiving process. This field can be omitted, in that case it is assumed the receiving process is on the same node. This field must by an integer data type.

- code:
  The code part of a message can contain an integer value. It can be used to give the receiving process some extra information about what kind of message you are sending or what you expect the receiving process to do with it. This field can be omitted, in which case it is assumed you are sending a string message to another pyTOOLS process. This field must by an integer data type.

- message:
  This field holds the actual messages you want to send. This message will always be of a string.

Syntax:

dss*.busSend(conn_hnd, message)*
- *conn_hnd*
  DSS connection handle
- *message*
  Python dictionary containing the message you want to send.

Returns:

None

Sending BUS/FAST process message example:

```
>>>
>>> import dss
>>>
>>> # Attach to BUS/FAST under the name 'sender'
>>> dss.busSetName('sender')
>>>
>>>> # Connect to DSS
```

```
>>> con = dss.connect()
>>>
>>> # Prepare a message. This message will be sent to process
>>> # called 'receiver'. Because the receiving process is on the
>>> # same FAST/TOOLS node we can omit the node field.
>>> message = {'process' : 'receiver',
...            'message' : 'HALLO RECEIVER'}
>>>
>>> # And finally, sent the message
>>> dss.busSend(con, message)
>>>
>>>
```

5.13.4 Sending a BUS/FAST message and wait for reply

Description:

Sometimes it can be convenient to send a message and then wait for an answer. For these occasions there is a function called busSendReceive. This function sends messages and waits a reply from the receiving process. If there is no answer within 60 seconds an error is raised. The message layout is the same as the layout of the busSend message. Please see section 5.13.3 for an explanation of the message layout. The reply is returned in the form of a Python dictionary. The contents of the reply may depend on the message code in your request, but will always be contained in a string.

Syntax:

reply_msg = ***dss.*** **busSendReceive***(conn_hnd, message)*

- *reply_msg*
  Dictionary containing the reply.
- *conn_hnd*
  DSS connection handle
- *message*
  Python dictionary containing the message you want to send.

After successful completion of the function the reply_msg dictionary will have the following contents:

```
reply_msg = { 'message' : 'Reply string'}
```

Returns:

A dictionary with the reply message

Sending BUS/FAST process message and waiting for reply example:

```
>>>
>>> import dss
>>>
>>> # Attach to BUS/FAST under the name 'sender'
>>> dss.busSetName('SENDER')
>>>
>>>> # Connect to DSS
>>> con = dss.connect()
>>>
>>> # Prepare a message. This message will be sent to process
>>> # called ‘receiver’. Because the receiving process is on the
>>> # same FAST/TOOLS node we can omit the node field.
>>> message = {'process' : 'receiver',
...             'message' : 'What’s the coolest language ? '}
>>>
>>> # And finally, sent the message
>>> answer = dss.busSendReiceive(con, message)
>>>
>>> # Now let see what the answer is
>>> print answer['message']
>>> 'Python of course ! '
```

5.13.5 Receiving BUS/FAST messages

Description:

Sending a BUS/FAST message can only be useful if the receiving process regularly checks if there are any messages waiting for him. This is exactly what the busReceive function does. If there is a BUS/FAST message waiting for you it will return a dictionary with the message. If there is no message this function will return ‘None’.

The.busReceive function can be called with an optional argument. If you pass the string “SYNC_WAIT” the function will not return until a BUS/FAST message was received. This can come in handy when writing a server process that just waits for messages to process.

If there is no message waiting and the “SYNC_WAIT” flag was not set the function will return None. If there was a message waiting we received a Python dictionary. The dictionary has the following fields:

```
        mesg = {'process' : 'sender',
                'node'    : 162,
                'reply'   : 0,
                'sys_type': 0,
                'code'    : 0,
                'message' : 'HELLO RECEIVER !' }
```

- process:
  Holds the name of the process that has send this message.

- node:
  This field contains node number of the sending process. If the sending process runs on the same FAST/TOOLS node this value will be 0.

- reply:
  This field indicates whether the sending process is waiting for a reply. If the value of reply is unequal 0 the sending process is waiting for a reply. In that case you should send a reply within 60 seconds using the busReply function. The value of the reply field must be included in your reply message.

- sys_type:
  This field indicates the operating system of the sending process. This is only of interest when the sending process requested an reply. In that case you must pass this value unaltered to the busReply function.
- code:
  The code part of a message can contain an integer value. It can be used by the sending process to give you some extra information about what kind of message you received or what you are expected to do with it.

- message:
  This field holds the actual messages that was send to you. This message will always be a string.

Syntax:

reply_msg = ***dss*.busReceive***(conn_hnd, flag)*

- *reply_msg*
  Dictionary containing the reply or 'None' if no message was waiting
- *conn_hnd*
  DSS connection handle
- *flag*
  String flag indication whether the function should return immediate if the message queue is empty or whether it should wait for a message. It you pass a string argument "SYNC_WAIT" busReceive will wait for a message to arrive.

Returns:

A dictionary with a BUS/FAST message or None

Waiting for BUS/FAST message example:

```
>>>
>>> import dss
>>>
>>> # Attach to BUS/FAST under the name 'sender'
>>> dss.busSetName('sender')
>>>
>>> # Connect to DSS
>>> con = dss.connect()
>>>
>>> message = dss.busReceive(con, 'SYNC_WAIT')
>>>
```

5.13.6 Reply to a BUS/FAST message

Description:

If you received a message where the value of the reply field is unequal to 0 this means the sending process is waiting for a reply. You should use the busReply function to send a reply to the waiting process.
The reply messages is stored in a Python dictionary. The dictionary has the following fields:

```
        reply_msg =  {'process' : 'sender',
                      'node'    : 162,
                      'reply'   : 5453,
                      'sys_type': 0,
                      'code'    : 0,
                      'message' : 'The answer is 42' }
```

- process:
  Holds the name of the process waiting for a reply.

- node:
  This field contains the FAST/TOOLS node number of the waiting process. If this field is omitted it is assumed that the waiting process is running on the same node.

- reply:
  This field contains the reply id that was included in the request message . The value of the reply field must be included unaltered in your reply message.

- sys_type:
  This field indicates the operating system of the waiting process. You received this value in the request message and you must pass this value unaltered to the busReply function.

- code:
  The code part of a message can contain an integer value. It can be used to give the receiving process some extra information about what kind of message you are sending or what you expect the receiving process to do with it. This field can be omitted, in which case it is assumed you are sending a string message to an other pyTOOLS process. This field must by an integer data type

- message:
  This field holds the actual reply messages you want to send. This message will always be of a string.

Syntax:

dss.**busReply***(conn_hnd, message)*

- *conn_hnd*
  DSS connection handle
- *message*
  Python dictionary containing the reply message you want to send to the waiting process.

Returns:

A dictionary with a BUS/FAST message or None

Replying a BUS/FAST request example:

```
>>>
>>> import dss
>>>
>>> # Attach to BUS/FAST under the name 'my_prog'
>>> dss.busSetName('my_prog')
>>>
>>> # Connect to DSS
>>> con = dss.connect()
>>>
>>> # Check if there is a message waiting for us
>>> mesg = dss.busReceive(con)
>>>
>>> # See there was as message waiting
>>> if mesg != None:
. . .       if mesg['reply'] != 0:
. . .            # Prepare a reply message, we can use the same
. . .            # message dictionary. Only the message field must
. . .            # be changed.
. . .            msg['message'] = 'The answer is: 42'
. . .            dss.busReply(con, msg)
>>>
```

```
>>>
```

5.13.7 Sending a BUS/FAST stop message

The correct way to stop any FAST/TOOLS process is to send it a stop request through BUS/FAST. If a process receives a stop request it should release any resource it holds and go exit. Of course pyTOOLS scripts, well-mannered as they are, are no exception to that rule. To send a stop request to a pyTOOLS server process called ‘SERVER’ from the command line you use the FAST/TOOLS command ‘durstp’

```
C:\>durstp -p server
```

When this request arrives at the server process it will raise an exception. Exception handling is explained in detail in chapter 6 of this manual. At this point all you have to realise is that exceptions in Python can have arguments, rather like functions. If a script receives a BUS/FAST stop request an exception is raised with 'PY_DSS_STOP_REQUEST’ as argument. The following example shows how to handle a stop request.

```
import pprint
import dss

# Connect to BUS/FAST as with process name server
dss.busSetName('SERVER')
con = dss.connect()

while 1:
    try:
        message = dss.busReceive(con, 'SYNC_WAIT')
        pprint.pprint(message)
    except dss.error, (code, msg):
        # Test exception code, if stop request leave the loop
        if code == 'PY_DSS_STOP_REQUEST':
            break
        else:
            print 'Error: ' + code + ' ' + msg

dss.close(con)
```

5.13.8 Getting BUS/FAST statistics

BUS/FAST keeps a list of runtime information that might be useful in your scripts. You can for example obtain a list of all processes connected to BUS/FAST and check their message queue statistics. Or you can check the quality of the connection to other FAST/TOOLS nodes in a multi node configuration. To be able to interpret the BUS/FAST statistics you need to have a reasonable knowledge of BUS/FAST. Please see the BUS/FAST manuals for a description of the BUS/FAST statistics.

The BUS/FAST statistics interface provides us with a snapshot of a running FAST/TOOLS system and is therefore called *busSnap*. This function can take a number of different snapshots of your FAST/TOOLS system.

The best way to demonstrate this is with an example. The ../examples/busfast directory of the pyTOOLS distribution contains a example script (snap.py) that demonstrates the use of the busSnap method.

The dss.busSnap method will return a dictionary containing the BUS/FAST statistics. There are number of different statistics categories and the fields in the dictionary will reflect the type of statistics you requested. The following statistics are available:

Category	Description
GENERAL	Takes a snapshot of general information.
QUEUE	Takes a snapshot of all process queues, their characteristics and performance.
NODE	Takes a snapshot of all know connected nodes, their characteristics and performance.
LOCK	Takes a snapshot of all locks, their characteristics and performance.

To obtain statistic information you first call the busSnap() function with the name of the category as parameter:

```
>>> import dss
>>> import pprint
>>> con = dss.connect()
>>> dss.busSnap('GENERAL')
```

This will tell BUS/FAST to collect the statistics but will not yet return any data. For that we need to call busSnap() again but this time with the parameter ‘FIRST’:

```
>>> general_info = dss.busSnap('FIRST')
>>> pprint.pprint(general_info)
{'common_high': 124928,
 'common_size': 2560000,
 'common_used': 108544,
 'customer': 'Yokogawa System Center Europe b.v.',
 'event_size': 500,
 'event_used': 4,
 'node_name': 'NODE_100',
 'node_nr': 100,
 'nodes': 0,
 'processes': 56,
 'queue_failed': 0,
 'queue_transf': 13381,
 'version': '8.3'}
>>>
```

The reason for this approach is that some of the statistics return more than one data elements. In that case BUS/FAST will create a list that can be read by the ‘FIRST’ and ‘NEXT’ parameters. The following example will print all the processes currently connected to BUS/FAST:

```
>>> import dss
>>> con = dss.connect()
>>> dss.busSnap('QUEUE')
>>> queue_info = dss.busSnap('FIRST')
>>> while(queue_info):
        print queue_info['name’]
        queue_info = dss.busSnap('NEXT')

CWIITM
DSSEVT01
CWICTH
DSSEVT00
FMFPRT
ITHSTX
……
……
```

When calling busSnap() with the parameter ‘FIRST’ it will return the first statistics element from the list. With the ‘NEXT’ parameter the statistics for the successive processes can be retrieved. Once the statistics for all processes have been returned the ‘NEXT’ call will return ‘None’.

5.14 Date and time functions

5.14.1 General

This subsection explains the use of the DSS specific date and time functions. Traditionally programming environments store date and time as the number of seconds that have been pasted since some arbitrary point in time, typically Jan 1st 1970. DSS uses a similar scheme but because FAST/TOOLS processes events on a millisecond basis the number of seconds can have a fractional the part, allowing date and time to be represented with a millisecond resolution.

Whenever you read a value from a DSS dataset that represents a date or time it will be presented as a floating-point value that represents the seconds since Jan 1st 1970 including fractions of a second. For example

```
May 20th 2003 at 10:55:00.445
```

is represented as `1053428100.445` which is the exact number of seconds, including the 445 milli-seconds that have passed since Jan 1st 1970. Luckily the pyTOOLS DSS package includes a number of functions the allows us the use date and time in a transparent way without the need to handle floating point representations.

5.14.2 Convert DSS format to string

Description:

This function converts DSS time format into a ASCII string. The function offers the possibility to pass a so-called ‘date format picture’ that determines the layout in which date and time are represented.
The formatting rules for the picture string must comply with are summarized in the following table. The examples all refer to the following string:

May 20th 2003 11:17:38.318

Picture string	Meaning	Example
YYYY	year	2003
YY	year	03
MMM	month	May
MM	month	20
M	month	5
DDD	day	Tue
DD	day	20
D	day	20
hh	hours	11
mm	minutes	17
ss	seconds	38
p	parts	of
pp	"	31
ppp	"	318
pppp	"	3180

For a complete abbreviations list for month and day formats see the table below.

Format	Result
MMM	Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec

Format	Result
DDD	Mon Tue Wed Thu Fri Sat Sun

If no format picture is passed the default layout will be used, being:

“DD-MMM-YY   hh:mm:ss”

Alternatively to constructing your own format picture you can make use of a number of predefined formats:

```
ANSI      = "DDD MMM DD hh:mm:ss YYYY"
JULIAN    = "MM/DD/YY hh:mm:ss"
EUROPE    = "DD/MM/YY hh:mm:ss"
GENERAL    = "DD-MM-YYYY hh:mm:ss"
```

Syntax:

date_string = **dss.dateString(***conn***,** *date, [date_format]***)**

- *date_string*
  String holding the date and time that was converted from DSS format
- *conn*
  DSS connection handle.
- date
  Date in DSS format
- *date_format*
  Optional date and time format string

Returns:

Date and time in ASCII string format

Convert DSS format example:

```
>>>
>>> import dss
>>> conn = dss.connect()
>>>
>>> dss.dateString(conn, 1053428100.445)
'20-MAY-03 10:55:00'
>>>
```

```
>>> dss.dateString(conn, 1053428100.445,"ANSI")
'TUE MAY 20 10:55:00 2003'
>>>
>>> dss.dateString(conn, 1053428100.445,"JULIAN")
'05/20/03 10:55:00'
>>>
>>> dss.dateString(conn, 1053428100.445,"DD-MMM-YY hh:mm:ss:ppp")
'20-MAY-03 10:55:00:445'
```

5.14.3 Absolute string format to DSS format

Description:

This function is used to convert an ASCII date/time string into DSS format. The date part of the string must have the following format:

```
dd-mm[m]-[yy]yy
```

Examples:

```
      20-May-03
      20-05-2003
```

The layout of the time part of the string must have the following format:

```
hh:mm[:ss[.[parts]]
```

Examples:

```
      10:50           =  10:50:00.000
      10:50:04        =  10:50:04.000
      10:50:04.003    =  10:50:04.003
      21:2:4.1        =  21:02:04.100
```

If either date or time is missing from a date/time format string the current time or date is used as a default.

Syntax:

date_dss = **dss.dateDSS(***conn***,** *date_string***)**

- *date_dss*
  Date and time in DSS format
- *conn*
  DSS connection handle.
- *date_string*
  String holding the date and time in ASCII format

Returns:

Date and time in DSS format

Absolute string format to DSS format example

```
>>>
>>> import dss
>>> conn = dss.connect()
>>>
```

```
>>> dss.dateDSS(conn,'10:50:04.003 20-May-03')
1053427804.003
>>>
>>> # If no date is included in the format string the current date is used
>>> dss.dateDSS(conn,'21:2:4.1')
1053464524.1
>>>
>>> # If no time is included in the format string the current time is used
>>> dss.dateDSS(conn, '20-May-03')
>>> 1053461440.0
>>>
```

5.14.4 Relative string format to DSS format

Description:

This function to allows you to enter a time relative to the current time. The date part of the string must have the following format:

```
dd-mm[m]-[yy]yy
```

Examples:

```
      20-May-03
      20-05-2003
```

The layout of the time part of the string must have the following format:

```
[[-]]days,][+ or -]hh:mm[:ss[.[parts]]
```

Examples:

```
      10:50                     = Today 10:50:00.000
      -1,10:50:04               = Yesterday 10:50:04.000
      1,10:50:04.003            = Tomorrow 10:50:04.003
      -5:30                     = Now minus 5 hours 30 minutes
      1:30                      = Now plus 1 hour 30 minutes
      20-05-2003 -1,10:50:04    = 19-05-2003 10:50:04
```

If either date or time is missing from a date/time format string the current time or date is used as a default.

Syntax:

date_dss = **dss.dateRelDSS(***conn, rel_date_string***)**

- *date_dss*
  *D*ate and time in DSS format
- *conn*
  DSS connection handle.
- *rel_date_string*
  String holding the relative date and time in ASCII format

Returns:

Date and time in DSS format.

Relative string format to DSS format example:

```
>>>
>>> import dss
>>> conn = dss.connect()
>>> dss.dateRelDSS(conn,"-1,10:50:04")
1053341404.0
>>>
>>> dss_time = dss.dateRelDSS(conn,"20-May-03 -1,10:50:04")
>>> dss.dateString(conn,dss_time,"JULIAN")
'05/19/03 10:50:04'
>>>
>>> dss_time = dss.dateRelDSS(conn,"19-Oct-02 5,10:50:04.003")
>>> dss.dateString(conn,dss_time,"ANSI")
'THU OCT 24 10:50:04 2002'
>>>
```

5.15 Time intervals

5.15.1 General

This subsection explains the use of the DSS interval type. Dataset fields of type interval can contain time intervals. This is for example used in the case of a report that should be generated periodically. Each record from the REPORT_DF dataset contains a field 'GENERATION_INTERVAL' of type 'interval'. These fields contain the interval at which the report must be generated.

DSS stores intervals in a variable of type DOUBLE in the form <number>.<interval>. For example 3 weeks is represented as 3.4 where the fractional part .4 represents weeks. Other supported intervals are depicted in the following table:

Interval period	Represented by
Second	0.0
Minute	0.1
Hour	0.2
Day	0.3
Week	0.4
Mouth	0.5
Year	0.6

Examples:

6 days	= 6.3
30 minutes	= 30.1
10 seconds	= 10.0

Because manipulating type DSS time interval in Python is extremely easy there are no API functions for it in the dss module. The following section explains how to use intervals in Python.

5.15.2 Working with DSS intervals

In this section we see how easy it is to use DSS intervals in Python. In the next example we will change the 'LIFE_TIME' of the ITEM_ACK report from 1 week to 3 days. For this we first have to open de REPORT_DF dataset:

```
>>> import dss
>>> con = dss.connect()
>>> fields = dss.getFieldNames(con, 'REPORT_DF')
>>> ds = dss.openDateset(con, 'REPORT_DF', fields, 'rudi')
```

Now we can read the record for the ITEM_ACK report: en get the contents of the 'LIFE_TIME' field:

```
>>> item_ack_rpt = dss.readEqual(con,dss,'ITEM_ACK')
>>> lifetime = str(item_ack_rpt['LIFE_TIME'])
>>> print lifetime
1.4
>>>
```

This value indicates a 'LIFE_TIME' of 1 week. To change it to 3 weeks we must set this value to 3.3.

```
>>> item_ack_rpt['LIFE_TIME'] = float('3.3')
>>> dss.updateRecord(con,ds,item_ack_rpt)
>>>
```

That’s it ! The ‘LIFE_TIME’ of the ITEM_ACK report has been changed from 1 week to tree days.

5.16 Item history

5.16.1 General

This subsection explains how the pyTOOLS module can be used to extract FAST/TOOLS item history. Each FAST/TOOLS item can be placed in a so-called “History group”. Inserting an item into a history group means FAST/TOOLS will store the values for that item according to the settings for that group. In the group settings you can for example configure how often the item values are to be stored and for how long they are to remain in the database.

Retrieving historical item data using de DSS-API is possible but slow and somewhat cumbersome. To provide a more efficient interface the pyTOOLS DSS module was extended with two functions that bypass DSS and directly interact with ITEM/FAST.

5.16.2 Get items in a history group

Description:

This function returns a list of the items that are in a history group.

Syntax:

Item_list = dss.getHisGroupItems(conn_hnd, group_name,[node])

- conn_hnd
  The DSS connection handle.
- group_name
  The name of the history group for which to retrieve the item names.
- *node*
  Optional parameter, the node parameter can be added when the history group is located on another FAST/TOOLS node in a distributed system.

Returns:

The Item names of the items in the history group as a list of strings.

Get items in a history group example:

This example shows how to get a list of all items in a history group. The history group is called ‘EVENT’

```
>>>
>>> import dss
>>> con = dss.connect()
>>> items = dss.getHisGroupItems(con, 'EVENT')
>>> print items
['PYTOOLS.TEST.REAL', 'PYTOOLS.TEST.INTEGER']
```

5.16.3 Get an interval of item history samples

Description:

This function returns an interval of item history for an item in a history group. The interval start and stop date are given as a DSS date objects. Please see section 5.14 on the use of the DSS date format. The parameter max_sample is the maximum number of history samples that will be returned.

As soon as the maximum number of samples was read the function will return, even if not all samples of the interval were read. This function will return a list of python tuples.

Each tuple in the list contains the following fields:

```
(date, value, status, option_bits, quality)
```

If all samples in the given interval were read before the maximum sample count was reached the last element in the returned list will be 'None'. This indicates no more sample need to be read for the interval.

If the last element is not 'None' you can take the date entry in the last tuple as the interval_starting for you next call to *dss.getItemHistory()* until you received all samples.

Syntax:

dss.getItemHistory(conn_hnd, item_name, group_name, interval_start, interval_end, max_sample [, node])

- conn_hnd
  The DSS connection handle.
- group_name
  The name of the history group for which to retrieve the item names.
- interval_start
  The start of the interval for which to retrieve item history samples. Given in DSS date format.
- interval_end
  The end of the interval for which to retrieve item history samples. Given in DSS date format.
- *max_sample*
  The maximal number of history sample the retrieve.
- *node*
  Optional parameter, the node parameter can be added when the history group is located on an other FAST/TOOLS node in a distributed system.

Returns:

A list of tuples

Get items history example:

This example shows how to get a list of history samples for an item.

```
>>>
>>> import dss
>>> con = dss.connect()

>>> # As interval start date we take 2 hours ago
>>> start = dss.dateRelDSS(con, "-02:00")
>>>
>>> # As interval end date we take 1 hours ago
>>> end = dss.dateRelDSS(con, "-01:00")
>>>
>>> his = dss.getItemHistory(con,'PYTOOLS.TEST.REAL','EVENT', start, end, 10)
>>>
[(1156688295.0, 12.0, 9, 8, 0), (1156688296.0, 13.0, 9, 8, 0), None]
```

5.17 Miscellaneous functions

5.17.1 General

The section contains useful functions that don’t fit in anywhere else.

5.17.2 Get error stack

Description:

This function gets all the messages on the UMH-stack and returns them in a list. The UMH messages are split in a code part, containing the UMH mnemonic, and a message part containing the message text. For more information on error handling and UMH-messages please see chapter 6.

Syntax:

error_list = **dss. getErrorStack(***conn_hnd*)

- *error_list*
  List holding the contents of the UMH stack
- *conn_hnd*
  DSS connection handle.

Returns:

A list of UMH messages.

5.17.3 Get Error stack example

```
>>> import dss
>>> conn = dss.connect()
>>> fields = dss.getFieldNames(conn, 'USER_DF')
>>> ds = dss.openDataset(conn,'USER_DF',fields,'ri')
>>>
>>> # read a record and try to insert that same record.
>>> # This will case an error.
>>> record = dss.readNext(conn,ds)
>>> dss.insertRecord(conn,ds,record)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
dss.error: ('VIS-W-DUPLICATE', 'Duplicate key')
>>>
>>> # get the contents of the UMH-stack.
>>> errors = dss.getErrorStack(conn)
>>>
>>> # print the UMH mnemonic part
>>> for err in errors:
...     print  err[0]
...
VIS-W-DUPLICATE
ISF-E-DUPKEY
ISF-I-WRITE
DSS-E-RECTHERE
FSL-I-MC_SERVER
>>>
>>> # print the UMH message part
>>> for err in errors:
```

```
...     print  err[1]
...
Duplicate key
Duplicate key not allowed
On write to isam file: C:\PROGRA~1\YOKOGA~1\FASTTO~1\tls\dat\dss_user.dat
The record already exists in data set USER_DF, or one of its fields has a value
which is used before
Message clustering: error coming from server process
>>>
```

5.17.4 Get DSS-module version

Description:

Use this function to check the version and build date of your pyTOOLS-DSS module

Syntax:

version = **dss. version()**

- *Version*
  String that holds the version and build date and time

Returns:

A string with version information.

Example:

```
>>> import dss
>>> dss.version()
'pyTOOLS DSSmodule 3.2  r2028   [May  5 2014, 10:37:54]'
```

5.17.5 Get paths to standard FAST/TOOLS directories

Description:

FAST/TOOLS files are organized into a number of default directories. The location of these directories is set during installation of FAST/TOOLS and may vary from project to project. The dss.busGetDirs() function will return a dictionary with the location of the default FAST/TOOLS directories.

Syntax:

dirs= **dss.busGetDirs()**

Returns:

A dictionary holding the location of the default FAST/TOOLS directories.

Example:

```
>>> import dss
>>> import pprint
>>> dirs = dss.busGetDirs()
>>> pprint.pprint(dirs)
{'TLS_AUX': 'C:\\PROGRA~1\\Yokogawa\\FASTTO~1\\tls\\aux\\',
 'TLS_COM': 'C:\\PROGRA~1\\Yokogawa\\FASTTO~1\\tls\\com\\',
 'TLS_DAT': 'C:\\PROGRA~1\\Yokogawa\\FASTTO~1\\tls\\dat\\',
 'TLS_DID': 'C:\\PROGRA~1\\Yokogawa\\FASTTO~1\\tls\\did\\',
 'TLS_EXE': 'C:\\PROGRA~1\\Yokogawa\\FASTTO~1\\tls\\exe\\',
 'TLS_HIS': 'C:\\PROGRA~1\\Yokogawa\\FASTTO~1\\tls\\his\\',
 'TLS_HLP': 'C:\\PROGRA~1\\Yokogawa\\FASTTO~1\\tls\\hlp\\',
 'TLS_LIB': 'C:\\PROGRA~1\\Yokogawa\\FASTTO~1\\tls\\lib\\',
 'TLS_LST': 'C:\\PROGRA~1\\Yokogawa\\FASTTO~1\\tls\\lst\\',
 'TLS_SAV': 'C:\\PROGRA~1\\Yokogawa\\FASTTO~1\\tls\\sav\\',
 'TLS_SUP': 'C:\\PROGRA~1\\Yokogawa\\FASTTO~1\\tls\\sup\\'}
>>>
>>>
```

5.17.6 Read FAST/TOOLS setup files

Description:

This function reads setup file parameters from setup files in the standard FAST/TOOLS setup file dir (/tls/sup). The contents of the setup file is returned as a two-dimensional list of strings with the following layout:

```
[["keyword_1", "parameter_1", "parameter_2",...."parameter_n"],
 ["keyword_2", "parameter_1", "parameter_2",...."parameter_n"],
 ["keyword_3", "parameter_1", "parameter_2",...."parameter_n"],
   .......
 ["keyword_n", "parameter_1", "parameter_2",...."parameter_n"]]
```

Syntax:

setup = dss.readSetup(conn_hnd, setupfile)

- conn_hnd
  The DSS connection handle.

- setupfile
  The name of the setup file without the .sup extension.

Returns:

The contents of the setup file is returned as a two-dimensional list of strings

Example:

This example read the contents of the dur.sup file in ../tls/sup.

```
>>> import dss
>>> import pprint
>>> con = dss.connect()
>>> setup = dss.readSetup(con, "dur")
>>> pprint.pprint(setup)
[['HOST_NODE', '0'],
 ['TIMEOUT_STEP', '1'],
 ['NCB_MAX_QUEUE', '200'],
 ['POOL_KBYTES', '5000'],
 ['SHM_START_ADDR', '0X50000000'],
 ['DUR_MAX_PROC', '250'],
 ['LANGUAGE', '0']]
```

```
>>>
>>>
```

5.17.7 Log UMH message

Description:

Log a message on the standard FAST/TOOLS message logger.

Syntax:

dss. logMessage (*conn_hnd*, *message_string*)

- *conn_hnd*
  DSS connection handle.
- *message_string*
  Test string you want to log on the standard FAST/TOOLS message logger. Maximal string length is 200.

Returns:

None

Example:

```
>>> import dss
>>> conn = dss.connect()
>>> dss.logMessage(conn, "Eat more fruit!")
```

CHAPTER 6

pyTOOLS Exception handling

Programmers don't die, they just GOSUB without RETURN.
-- Oleg Broytmann

Like most modern object oriented languages Python has built-in support for exception handling. Chapter eight of the standard Python tutorial gives an excellent explanation on this subject. This section will concentrate mainly on the way pyTOOLS exceptions are handled.

6.1 Handling exceptions

The following pieces of code demonstrate how Python's exception handling works. In the first example there is no exception handling:

```
>>> count = 5
>>> while 1:
...     print 1.0/count
...     count = count -1
...
0.2
0.25
0.333333333333
0.5
1.0
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
ZeroDivisionError: float division
```

When an attempt is made to divide by zero the Python interpreter raises a `ZeroDivisionError` exception. If this exception is not handled by the program the interpreter stops execution and the exception message is printed on the screen.

The second example makes use of exception handling:

```
>>> count = 5
>>> while 1:
...     try:
...         print 1.0/count
...         count = count - 1
...     except ZeroDivisionError:
...         print "Oops, you divided by zero"
...         break
...
0.2
0.25
0.333333333333
0.5
1.0
Oops, you divided by zero
>>>
```

If an exception occurs in the try clause (the statements between the `try` and the `except` keywords) the Python interpreter checks if there are any `except` clauses defined for that particular exception. In the except clause you put the code you want to be executed in case the exception occurs. The `try` clause can be followed by as many except clauses, as you like. Python has a large number of standard exception types that are all listed and explained in the Python library reference (filename: lib.pdf).

6.1.1 Exception arguments

In Python exceptions can have arguments. Consider the following:

```
>>> try:
...     1/0
... except ZeroDivisionError, arg:
...     print 'Run-time error:', arg
...
Run-time error: integer division or modulo
>>>
```

The `ZeroDivisionError` has one an argument that is assigned to the variable `arg`. The use of exception arguments is optional.

6.2 pyTOOLS exceptions

As we have seen in the previous section Python has a range of predefined exceptions. Apart from that each program or module can add its own exceptions. The pyTOOLS-DSS module introduces a single exception, it is simply called *dss.error*. The following example demonstrates the use of the dss.error exception.

```
>>> import dss
>>> con = dss.connect()
>>> fields = dss.getFieldNames(con, 'ITEM_DF')
>>> ds = dss.openDataset(con, 'ITEM_DF', fields, 'r')
>>>
>>> while 1:
...     item = dss.readNext(con,ds)
...
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
dss.error: ('DSS-E-EOS', 'Reached end of data set')
>>>
```

The dss.error exception has two arguments, an error-code, in this case: *DSS-E-EOS* and an error-message in this case: *Reached end of data set*. If you are familiar with the FAST/TOOLS UMH-message handler you will recognise that these are the code and the message from the UHM-message that was created when DSS couldn’t read beyond the end of the ITEM_DF data set.

There are two kinds of dss.error exceptions. Exceptions can be based on an UMH-message coming from FAST/TOOLS or they can originate from the DSS-module itself.

6.2.1 FAST/TOOLS exceptions

A FAST/TOOLS exception occurs when a DSS-API routine does not succeed. In that case the pyTOOLS DSS-module will take the most recent message on the UMH-stack and pass the message code and text as arguments to the exception. For example:

```
>>> import dss
>>>
>>> con = dss.connect()
>>> fields = dss.getFieldNames(con, 'ITEM_DF')
>>> ds = dss.openDataset(con, 'ITEM_DF', fields, 'r')
>>>
>>> while 1:
...     try:
...         item = dss.readNext(con,ds)
...         print item['NAME']
...     except dss.error, (code, message):
...         if code == 'DSS-E-EOS':
...             print message
...         break
...
INSTAL.UNIT.TAG1
INSTAL.UNIT.TAG2
INSTAL.UNIT.TAG3
INSTAL.UNIT.TAG4
INSTAL.UNIT.TAG5
INSTAL.UNIT.TAG6
INSTAL.UNIT.TAG7
Reached end of data set
>>>
>>>
```

This example also shows that exceptions aren't always a bad thing. In fact the only way to check if the readNext to has reached the end of the data set is to catch the 'DSS-E-EOS' error code.

Tip: If you want to inspect the complete UMH-stack after an error occurred you can use the function *dss.getErrorStack*, which returns all errors on the stack.

6.2.2 pyTOOLS-DSS module exceptions

Apart from the exceptions that are generated by FAST/TOOLS UMH error messages the pyTOOLS-DSS-module can raise some exceptions by itself. It does for example some checking on the data types of function arguments before handing then over to FAST/TOOLS. If a function argument is of the wrong data type a *dss.error* exception is raised. The code argument of a DSS-module exception is always starts with "PY_DSS_…". and the message argument holds a description of what when wrong. In the following example the user tries to connect twice to DSS, which is not possible:

```
>>> a = dss.connect()
>>> b = dss.connect()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
dss.error: ('PY_DSS_ALREADY_CONN', 'Already connected to DSS')
>>>
```

The following table shows the possible exceptions that the pyTOOLS DSS module can raise in pyTOOLS

pyTOOLS DSS error code:	**Error message:**
PY_DSS_ALREADY_CONN	Already connected to DSS
PY_DSS_DS_IS_CLOSED	Data set is closed.
PY_DSS_EVT_NOTIFY_DEF	Event notification already defined for dataset.
PY_DSS_FLT_DEF	Dataset has already a filter defined.
PY_DSS_FLT_NOT_ACTIV	No filter active for dataset.
PY_DSS_FLT_NOT_FOUND	Filter not found.
PY_DSS_FLT_TOO_LONG	Filter expression too long (max = 252).
PY_DSS_MISG_IDX	Missing field in index:
PY_DSS_NO_CHAN_OPEN	No event channel open for data set.
PY_DSS_NO_EVT_DEF	No event notification defined for data set.
PY_DSS_NOT_CONN	Not connected to DSS
PY_DSS_UNK_ERR	Function returned wrong or unknown error code.
PY_DSS_UNK_FLD	Unknown field
PY_DSS_UNK_FLD_TYPE	Unknown or wrong data type for field

CHAPTER 7

The pyTOOLS Data structures

If the map and the terrain disagree,
trust the terrain.
-- Swiss army aphorism

This section describes the pyTOOLS data structures. The pyTOOLS functions make use of two data types of structures, *lists* and *dictionaries*. The next two paragraphs will explain what lists and dictionaries are and how they can be used. Most of the examples are taken from the Python tutorial where you find much more information.

7.1 List

Python knows a number of *compound* data types, used to group together other values. The most versatile is the *lis*t, which can be written as a list of comma-separated values (items) between square brackets. List items need not all have the same type.

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> a
['spam', 'eggs', 100, 1234]
```

Like string indices, list indices start at 0, and lists can be sliced, concatenated and so on:

```
>>> a[0]
'spam'
>>> a[3]
1234
>>> a[-2]
100
>>> a[1:-1]
['eggs', 100]
>>> a[:2] + ['bacon', 2*2]
['spam', 'eggs', 'bacon', 4]
>>> 3*a[:3] + ['Boe!']
['spam', 'eggs', 100, 'spam', 'eggs', 100, 'spam', 'eggs', 100, 'Boe!']
```

7.2 Dictionaries

Another useful data type built into Python is the *dictionar*y. Dictionaries are sometimes found in other languages as “associative memories” or “associative arrays”. Unlike lists, which are indexed by numbers, dictionaries are indexed by *key*s.

It is best to think of a dictionary as an unordered set of *key: value* pairs, with the requirement that the keys are unique (within one dictionary). A pair of braces creates an empty dictionary: {}. Placing a comma-separated list of key:value pairs within the braces adds initial key:value pairs to the dictionary; this is also the way dictionaries are written on output. The main operations on a dictionary are storing a value with some key and extracting the value given the key. It is an error to extract a value using a non-existent key.

The `keys()` method of a dictionary object returns a list of all the keys used in the dictionary, in random order (if you want it sorted, just apply the `sort()` method to the list of keys). To check whether a single key is in the dictionary, use the `has_key()` method of the dictionary. Here is a small example using a dictionary:

```
>>> tel = {’jack’: 4098, ’sape’: 4139}
>>> tel[’guido’] = 4127
>>> tel
{’sape’: 4139, ’guido’: 4127, ’jack’: 4098}
>>> tel[’jack’]
4098
>>> del tel[’sape’]
>>> tel[’irv’] = 4127
>>> tel
{’guido’: 4127, ’irv’: 4127, ’jack’: 4098}
>>> tel.keys()
[’guido’, ’irv’, ’jack’]
>>> tel.has_key(’guido’)
1
```

7.3 List in pyTOOLS

Lists in pyTOOLS are used to hold names of fields or data set. The list that the different DSS-module functions return or taken as argument are all demonstrated in the examples in this document.

7.4 Dictionaries in pyTOOLS

The pyTOOLS-DSS module uses dictionaries as compound data type. Their *key:value* concept makes then very useful when working with tables, such as data sets. If you come from a ‘C’ background it is perhaps best to think of dictionaries as a kind of structures. PyTOOLS uses dictionaries for the following data:

- The DSS connection handle
- The data set handle
- The data set record

7.5 The DSS connection handle

The function *dss.connect* opens a connection to the DSS server. It returns a dictionary with the following layout:

```
{'conn_hnd': 2717908995L,
 'ds_table': {},
 'password': 'SYSMAN',
 'user': 'SYSMAN'}
```

The ‘user’ and ‘password’ field are only present after a logon. When the connection to the DSS server is closed the connection handle dictionary is made empty. When a dataset is opened a reference to the dataset handle is placed in the `'ds_table'` dictionary. This is necessary because the pyTOOLS-DSS module needs a central store to reach all datasets in case the program requests event notification

7.6 The DSS data set handle

The function *dss.openDataset* opens a data set and returns a dataset connection handle object. The connection handle is in fact a dictionary with the following layout:

```
{'fields':
      {'DESCRIPTION': {'buffer': <PyCObject object at 018B15A0>,
                       'length': 80,
                       'type': 100},
      'INSTALL':      {'buffer': <PyCObject object at 018B1580>,
                       'length': 16,
                       'type': 9},
      'NAME':         {'buffer': <PyCObject object at 018B1D90>,
                       'length': 32,
                       'type': 9},
      'STATUS':       {'buffer': <PyCObject object at 018B2F40>,
                       'length': 4,
                       'type': 1},
      'UNIT':         {'buffer': <PyCObject object at 018B1600>,
                       'length': 16,
                       'type': 9}},
 'fld_set_hnd': 2717908993L,
 'index': 'NAME',
 'name': 'UNIT_DF',
 'prime_index': 'NAME',
 'set_hnd': 2717908993L}
```

This dictionary has the following key's:

- *fields*
  This key holds a dictionary object for each field in the data set. For each field there is a data buffer, the field length and the filed type.
- *fld_set_hnd*
  Handle to the DSS field set. For internal use only.
- *index*
  The currently active index.
- *name*
  The name of the data set.
- *prime_index*
  This is the default index that is set when the data set is opened.
- *set_handle*
  Handle to the DSS data set. For internal use only.

When event notification is requested for a dataset an additional dictionary is added to the dataset connection handle to hold event notification information example:

```
dss.datasetEventNotify(conn, ds ,'u')
```

If this request is success full it will be stored in the dataset connection handle object:

```
{'events':
      {'evt_hnd': 2717908993L,
       'records': {},
       'dataset': {'mode': 'u'}},
'fields':
      {'DESCRIPTION': {'buffer': <PyCObject object at 018B15A0>,
                       'length': 80,
                       'type': 100},
      'INSTALL':      {'buffer': <PyCObject object at 018B1580>,
                       'length': 16,
                       'type': 9},
      'NAME':         {'buffer': <PyCObject object at 018B1D90>,
                       'length': 32,
                       'type': 9},
      'STATUS':       {'buffer': <PyCObject object at 018B2F40>,
                       'length': 4,
                       'type': 1},
      'UNIT':         {'buffer': <PyCObject object at 018B1600>,
                       'length': 16,
                       'type': 9}},
 'fld_set_hnd': 2717908993L,
 'index': 'NAME',
 'name': 'UNIT_DF',
 'prime_index': 'NAME',
 'set_hnd': 2717908993L}
```

The events dictionary holds the following data:

- *evt_hnd*
  This key holds an event handle. Messages in the DSS event queue contain a event handle number to uniquely identify the requester.
- *records*
  Contains an entry for each data set record for with event notification was requested.
- *dataset*
  If event notification was requested on dataset level this dictionary contains the type of notification

7.6.1 What can I do with it?

First of all you should ***never*** change any of the values in de data set handle object. They are used by the pyTOOLS C-routine in the DSS extension module and changing any values will very likely crash your application! What you can do is read information out of it. You could get the currently active index or the length of a particular dataset field. Example:

```
>>> import dss
>>>
>>> conn = dss.connect()
>>> fields = dss.getFieldNames(conn, 'UNIT_DF')
>>> ds_hnd = dss.openDataset(conn, 'UNIT_DF', fields, 'r')
>>>
>>> ds_hnd['index']
'NAME'
>>> ds_hnd['fields']['NAME']['length']
32
>>>
```

If you want to inspect to contents of any complex data structure you can use the *pretty-print* module pprint that is part of the Python distribution.

```
>>> import pprint
>>> pprint.pprint(ds_hnd)
{'fields': {'DESCRIPTION': {'buffer': <PyCObject object at 00804A80>,
                            'length': 80,
                            'type': 100},
            'INSTALL': {'buffer': <PyCObject object at 00804A60>,
                        'length': 16,
                        'type': 9},
            'NAME': {'buffer': <PyCObject object at 008048D0>,
                     'length': 32,
                     'type': 9},
            'STATUS': {'buffer': <PyCObject object at 00800010>,
                       'length': 4,
                       'type': 1},
            'UNIT': {'buffer': <PyCObject object at 00804AE0>,
                     'length': 16,
                     'type': 9}},
 'fld_set_hnd': 2717908993L,
 'index': 'NAME',
 'name': 'UNIT_DF',
 'prime_index': 'NAME',
 'set_hnd': 2717908993L}
>>>
```

CHAPTER 8

What Now?

"Wot? No quote?"
-- Guido van Rossum

I am sure by now you just can't wait to get started with Python. This manual has told you just about every thing there is to know about the pyTOOLS-DSS extension but it barely touched upon the possibilities, power and beauty of the Python programming language itself.

To become really excited you should read the Python tutorial and skim through the library reference, both come with the standard documentation. The library reference contains a huge amount of modules ready to use in your scripts.

Python's home on the internet is http://www.python.org . Form here you can download the Python interpreter, source code and lots of documentation. It has links to the best Python related web sites and contains a large list of Frequently Asked Questions (FQA) http://www.python.org/doc/FAQ.html

To get in contact with other Pythoneers, to ask questions and to learn a lot about Python and programming in general you can have a look at the Python newsgroup comp.lang.python.

Enjoy!