# *Script Manager* Training Module

## Revision History

| Version | Date | Revision Description |
|---|---|---|
| 0.1 | 10/12/2014 | Initial draft |
| | | |
| | | |
| | | |

# Contents

# 1. Introduction

This document is part of training modules for the Nile Basin Decision Support System (DSS). These modules are developed for use in classroom training that is given to Nile Basin countries and as a self-learning training material that will be made available as part of the DSS helpdesk and knowledgebase.

## 1.1. Purpose

The purpose of this document is to provide a tutorial on the DSS Script Manager. The tutorial starts with the basics and progressively increases in complexity.

## 1.2. Module pre-requisites

The following prerequisites are needed before taking this tutorial:
Software prerequisites: The Mike by DHI version 2014 and the DSS version 2.0 have to be installed.
User prerequisites: User is expected to be familiar with the DSS User Interface basics.

## 1.3. Expectations

Upon successful completion of the lessons, exercises and review questions in this document, you will be familiar with most of the Script Manager functionalities.

## 1.4. Conventions

The following conventions are followed in this document:
means a tip for the user

 means important information

## 1.5. Module data

Files that are needed for this module are located at the **..\ScriptsExp\data** folder.

## 1.6. Links to additional resources

In addition to the information presented in this module, below are links to additional resources that you can access to obtain further information on the following:

- Script Manager:
  - The DSS help file accessible by clicking on the  button
- Iron Python scripting language:
  - http://ironpython.net/

## 1.7. Problem Reporting Instructions

This document will be updated regularly. Therefore, it is highly recommended to report any spotted problem to helpdesk@nilebasin.org so it can be corrected in future versions. When reporting the problem, you are kindly requested to provide the following:

- Document title
- Document version
- Page number where the problem was spotted
- A description of the problem

# 2. Lessons

In this section the following lessons (with exercises) are included:

- General: This lesson introduces you to script definition in general and within the DSS, uses of scripts in the DSS. It then gives an overview of the 'Iron Python' the scripting language used in the DSS.
- Iron Python primer: This lesson gives a basic explanation of language components and their syntax.
- Script Manager basics: This lesson introduces you to the Script Manager components, how scripts are stored in the DSS, DSS script types and to some basic tasks such as activating the manager.
- Creating simple scripts: This lesson shows you how you can create, debug and save a simple script in the DSS.
- Handling changes and metadata: This lesson introduces you to the change log and metadata sections of each script. It also shows how they can be used.
- Creating complex scripts: This lesson shows you how you can create, debug and save a complex script in the DSS.
- Predefined scripts: This lesson gives an overview of the DSS predefined scripts. It also shows you how you can expand the predefined indicators
- Advanced scripting: This lesson introduces you to two advanced scripting topics, namely, using the DSS Application Programming Interface in scripts (including accessing DSS objects such as time series, GIS layers, scenarios and spreadsheets) and using DSS tools in a script.

  After completing the lessons and exercises in this section you will be able to use the Scripts Manager to manage scripts within the DSS.

## 2.1. General

### Introduction

This lesson introduces you to scripting in general and within the DSS, and to uses of scripts in the DSS. It then gives an overview of the 'Iron Python', the scripting language used in the DSS. If you are familiar with those definitions and concepts you can skip this and move to the next lesson.

Topics covered in this lesson:

- What is a script? And what are its uses in the DSS?
- an overview of the 'Iron Python' the scripting language

Lesson objectives:

After completing this lesson, you will be familiar with the following:

- Script- concepts and uses in the DSS.
- The 'Iron Python' scripting language.

### What is a script?

A script is a series of instructions that are written using a scripting language to typically automate repetitive tasks. These instructions are interpreted or carried out by another program (interpreter) rather than directly by the computer processor (as a compiled program is). To give an example, The DSS is a compiled program which runs directly by the computer processor. If you write a script within the DSS, you don't need to compile it and run separately. It can run within the DSS which will interpret it line by line. In this case the script instructions are passed to the computer processor via the DSS (i.e. the interpreter is part of the compiled DSS.

### What are the uses of Scripts in the DSS?

In the DSS, scripts can be used to for the following various reasons:

- Automate repetitive tasks. Imagine you have daily task of checking daily rainfall data records of a number of catchment gauges. To do this, you can write a script to import and check this data using the DSS tools.
- Calculate the value of an indicator. For example, if you want to calculate the evaporation losses from a reservoir, you can write a script that processes

the evaporation time series of this reservoir (i.e. using the reservoir model results) and then calculates the total evaporation losses from this reservoir.

- Create customized functionality in the DSS such as creating other Managers Tools, or model Adapters.

## What is the 'Iron Python' scripting language?

Iron Python is the scripting language of the DSS. It is an open-source implementation of the Python programming language[1]. Iron Python is integrated within the Microsoft .NET Framework and can use both the .NET Framework and Python libraries. Other .NET languages can also use Iron Python code. It is considered as an excellent addition to the .NET Framework, providing Python developers with the power of the .NET framework. Existing .NET developers can also use Iron Python as a fast and expressive scripting language for embedding, testing, or writing new applications. For more details about the language see the IronPython primer section

## Review Questions

1. What is a script?
2. What are the uses of scripts in the DSS?

[1] See more details at http://python.org/

## Answers

1. A script is a series of instructions that are written using a scripting language to typically automate repetitive tasks.
2. In the DSS, scripts can be used to for the following various reasons:
    - Automate repetitive tasks.
    - Calculate the value of an indicator.
    - Create customized functionality in the DSS such as creating other Managers Tools, or model Adapters.

## 2.2. IronPython primer

### Introduction

This primer will attempt to teach you Python[2]. It will just show you some basic concepts to start you off. It assumes that you are already familiar with programming and will, therefore, skip most of the non-language-specific material. The important keywords will be highlighted so you can easily spot them. Also, pay attention because, due to the nature of this tutorial, some things will be introduced directly in code and only briefly commented on. This primer also assumes that you have already installed Python on your computer.

Lesson objectives:
By the end of this lesson, it is anticipated that you will be familiar with the Iron Python language components and their syntax.

### Lesson pre-requisites

You have to be familiar with the programming basics to take this lesson.

### Properties

Python is strongly typed (i.e. types are enforced), dynamically, implicitly typed (i.e. you don't have to declare variables), case sensitive (i.e. var and VAR are two different variables) and object-oriented (i.e. everything is an object) scripting language.

### Getting help

Help in Python is always available right in the interpreter. If you want to know how an object works, all you have to do is call `help(<object>)`.Also useful are dir(), which shows you all the object's methods,
and `<object>.__doc__`, which shows you its documentation string:

```
>>> help(5)
Help on int object:
```

[2] Python and Iron Python are very similar but not identical

```
(etc etc)

>>> dir(5)
['__abs__', '__add__', ...]

>>> abs.__doc__
'abs(number) -> number

Return the absolute value of the argument.'
```

**Syntax**

Python has no mandatory statement termination characters and blocks are specified by indentation. Indent to begin a block, dedent to end one. Statements that expect an indentation level end in a colon (:). Comments start with the pound (#) sign and are single-line, multi-line strings are used for multi-line comments. Values are assigned (in fact, objects are bound to names) with the _equals_ sign ("="), and equality testing is done using two _equals_ signs ("==").
You can increment/decrement values using the += and -= operators respectively by the right-hand amount. This works on many datatypes, strings included. You can also use multiple variables on one line. For example:

```
>>> myvar = 3
>>> myvar += 2
>>> myvar
5
>>> myvar -= 1
>>> myvar
4
"""This is a multiline comment.
The following lines concatenate the two strings."""
>>> mystring = "Hello"
>>> mystring += " world."
>>> print mystring
Hello world.
# This swaps the variables in one line(!).
# It doesn't violate strong typing because values aren't
# actually being assigned, but new objects are bound to
# the old names.
>>> myvar, mystring = mystring, myvar
```

### Data sturctures

The data structures available in python are lists, tuples and dictionaries. Sets are available in the `sets` library (but are built-in in Python 2.5 and later). Lists are like one-dimensional arrays (but you can also have lists of other lists), dictionaries are associative arrays (a.k.a. hash or look-up tables) and tuples are immutable one-dimensional arrays (Python "arrays" can be of any type, so you can mix e.g. integers, strings, etc in lists/dictionaries/tuples). The index of the first item in all array types is 0. Negative numbers count from the end towards the beginning, -1 is the last item. Variables can point to functions. Note that lists use square brackets [ ], tuples use parentheses ( ) while dictionaries use braces { }. The usage is as follows:

```
>>> sample = [1, ["another", "list"], ("a", "tuple")]
>>> mylist = ["List item 1", 2, 3.14]
>>> mylist[0] = "List item 1 again" # We're changing the item.
>>> mylist[-1] = 3.21 # Here, we refer to the last item.
>>> mydict = {"Key 1": "Value 1", 2: 3, "pi": 3.14}
>>> mydict["pi"] = 3.15 # This is how you change dictionary
values.
>>> mytuple = (1, 2, 3)
>>> myfunction = len
>>> print myfunction(mylist)
3
```

You can access array ranges using a colon (:). Leaving the start index empty assumes the first item, leaving the end index assumes the last item. Negative indexes count from the last item backwards (thus -1 is the last item) like so:

```
>>> mylist = ["List item 1", 2, 3.14]
>>> print mylist[:]
['List item 1', 2, 3.1400000000000001]
>>> print mylist[0:2]
['List item 1', 2]
>>> print mylist[-3:-1]
['List item 1', 2]
>>> print mylist[1:]
[2, 3.14]
```

```
# Adding a third parameter, "step" will have Python step in
# N item increments, rather than 1.
# E.g., this will return the first item, then go to the third
and
# return that (so, items 0 and 2 in 0-indexing).
>>> print mylist[::2]
['List item 1', 3.14]
```

### Strings

Strings can use either single or double quotation marks, and you can have quotation marks of one kind inside a string that uses the other kind (i.e. "He said 'hello'." is valid). Multiline strings are enclosed in triple double (or single) quotes (""""). Python supports Unicode out of the box, using the syntax u"This is a unicode string". To fill a string with values, you use the % (modulo) operator and a tuple. Each %s gets replaced with an item from the tuple, left to right, and you can also use dictionary substitutions, like so:

```
>>>print "Name: %s\
Number: %s\
String: %s" % (myclass.name, 3, 3 * "-")
Name: Poromenos
Number: 3
String: ---

strString = """This is
a multiline
string."""

# WARNING: Watch out for the trailing s in "%(key)s".
>>> print "This %(verb)s a %(noun)s." % {"noun": "test",
"verb": "is"}
This is a test.
```

### Flow control statements

Flow control statements are `if`, `for`, and `while`. There is no `select`; instead, use if. Use `for to` enumerate through members of a list. To obtain a list of numbers, use `range(<number>)`. These statements' syntax is thus:

```
rangelist = range(10)
```

```
>>> print rangelist
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
for number in rangelist:
    # Check if number is one of
    # the numbers in the tuple.
    if number in (3, 4, 7, 9):
        # "Break" terminates a for without
        # executing the "else" clause.
        break
    else:
        # "Continue" starts the next iteration
        # of the loop. It's rather useless here,
        # as it's the last statement of the loop.
        continue
else:
    # The "else" clause is optional and is
    # executed only if the loop didn't "break".
    pass # Do nothing

if rangelist[1] == 2:
    print "The second item (lists are 0-based) is 2"
elif rangelist[1] == 3:
    print "The second item (lists are 0-based) is 3"
else:
    print "Dunno"

while rangelist[1] == 1:
    pass
```

**Functions**

Functions are declared with the "def" keyword. Optional arguments are set in the function declaration after the mandatory arguments by being assigned a default value. For named arguments, the name of the argument is assigned a value. Functions can return a tuple (and using tuple unpacking you can effectively return multiple values). Lambda functions are ad hoc functions that are comprised of a single statement. Parameters are passed by reference, but immutable types (tuples, ints, strings, etc) *cannot be changed*. This is because only the memory location of the item is passed, and binding another object to a variable discards the old one, so immutable types are replaced. For example:

```
# Same as def funcvar(x): return x + 1
funcvar = lambda x: x + 1
>>> print funcvar(1)
2

# an_int and a_string are optional, they have default values
# if one is not passed (2 and "A default string",
respectively).
def passing_example(a_list, an_int=2, a_string="A default
string"):
    a_list.append("A new item")
    an_int = 4
    return a_list, an_int, a_string

>>> my_list = [1, 2, 3]
>>> my_int = 10
>>> print passing_example(my_list, my_int)
([1, 2, 3, 'A new item'], 4, "A default string")
>>> my_list
[1, 2, 3, 'A new item']
>>> my_int
10
```

### Classes

Python supports a limited form of multiple inheritance in classes. Private variables and methods can be declared (by convention, this is not enforced by the language) by adding at least two leading underscores and at most one trailing one (e.g. "__spam"). We can also bind arbitrary names to class instances. An example follows:

```
class MyClass(object):
    common = 10
    def __init__(self):
        self.myvariable = 3
    def myfunction(self, arg1, arg2):
        return self.myvariable

    # This is the class instantiation
>>> classinstance = MyClass()
>>> classinstance.myfunction(1, 2)
3
# This variable is shared by all classes.
```

```
>>> classinstance2 = MyClass()
>>> classinstance.common
10
>>> classinstance2.common
10
# Note how we use the class name
# instead of the instance.
>>> MyClass.common = 30
>>> classinstance.common
30
>>> classinstance2.common
30
# This will not update the variable on the class,
# instead it will bind a new object to the old
# variable name.
>>> classinstance.common = 10
>>> classinstance.common
10
>>> classinstance2.common
30
>>> MyClass.common = 50
# This has not changed, because "common" is
# now an instance variable.
>>> classinstance.common
10
>>> classinstance2.common
50

# This class inherits from MyClass. The example
# class above inherits from "object", which makes
# it what's called a "new-style class".
# Multiple inheritance is declared as:
# class OtherClass(MyClass1, MyClass2, MyClassN)
class OtherClass(MyClass):
    # The "self" argument is passed automatically
    # and refers to the class instance, so you can set
    # instance variables as above, but from inside the class.
    def __init__(self, arg1):
        self.myvariable = 3
        print arg1

>>> classinstance = OtherClass("hello")
hello
```

```
>>> classinstance.myfunction(1, 2)
3
# This class doesn't have a .test member, but
# we can add one to the instance anyway. Note
# that this will only be a member of classinstance.
>>> classinstance.test = 10
>>> classinstance.test
10
```

### Exceptions

Exceptions in Python are handled with try-except [exceptionname] blocks:

```
def some_function():
    try:
        # Division by zero raises an exception
        10 / 0
    except ZeroDivisionError:
        print "Oops, invalid."
    else:
        # Exception didn't occur, we're good.
        pass
    finally:
        # This is executed after the code block is run
        # and all exceptions have been handled, even
        # if a new exception is raised while handling.
        print "We're done with that."

>>> some_function()
Oops, invalid.
We're done with that.
```

### Importing

External libraries are used with the `import [libname]` keyword. You can also use `from [libname] import [funcname]` for individual functions.

Here is an example:

```
import random
from time import clock

randomint = random.randint(1, 100)
>>> print randomint
```

```
64
```

### File I/O

Python has a wide array of libraries built in. As an example, here is how serializing (converting data structures to strings using the `pickle` library) with file I/O is used:

```
import pickle
mylist = ["This", "is", 4, 13327]
# Open the file C:\\binary.dat for writing. The letter r
before the
# filename string is used to prevent backslash escaping.
myfile = open(r"C:\\binary.dat", "w")
pickle.dump(mylist, myfile)
myfile.close()

myfile = open(r"C:\\text.txt", "w")
myfile.write("This is a sample string")
myfile.close()

myfile = open(r"C:\\text.txt")
>>> print myfile.read()
'This is a sample string'
myfile.close()

# Open the file for reading.
myfile = open(r"C:\\binary.dat")
loadedlist = pickle.load(myfile)
myfile.close()
>>> print loadedlist
['This', 'is', 4, 13327]
```

### Miscellaneous

- Conditions can be chained. `1 < a < 3` checks that a is both less than 3 and greater than 1.
- You can use `del` to delete variables or items in arrays.
- List comprehensions provide a powerful way to create and manipulate lists. They consist of an expression followed by a `for` clause followed by zero or more `if` or `for` clauses, like so:

```
>>> lst1 = [1, 2, 3]
>>> lst2 = [3, 4, 5]
>>> print [x * y for x in lst1 for y in lst2]
[3, 4, 5, 6, 8, 10, 9, 12, 15]
>>> print [x for x in lst1 if 4 > x > 1]
[2, 3]
# Check if a condition is true for any items.
# "any" returns true if any item in the list is true.
>>> any([i % 3 for i in [3, 3, 4, 4, 3]])
True
# This is because 4 % 3 = 1, and 1 is true, so any()
# returns True.

# Check for how many items a condition is true.
>>> sum(1 for i in [3, 3, 4, 4, 3] if i == 4)
2
>>> del lst1[0]
>>> print lst1
[2, 3]
>>> del lst1
```

1. Global variables are declared outside of functions and can be read without any special declarations, but if you want to write to them you must declare them at the beginning of the function with the "global" keyword, otherwise Python will bind that object to a new local variable (be careful of that, it's a small catch that can get you if you don't know it). For example:

```
number = 5

def myfunc():
    # This will print 5.
    print number

def anotherfunc():
    # This raises an exception because the variable has not
    # been bound before printing. Python knows that it an
    # object will be bound to it later and creates a new,
local
    # object instead of accessing the global one.
    print number
    number = 3

def yetanotherfunc():
    global number
```

```
    # This will correctly change the global.
    number = 3
```

## Review Questions

1. What is the differences between a list, a dictionary and a tuple?
2. Does Python have a “select” statement?
3.

## Answers

## 2.3. Script Manager basics

### Introduction

This lesson introduces you to the Script Manager basics.

Topics covered in this lesson:

- Script Manager components
- How scripts are stored in the DSS
- DSS script types
- Basic tasks such as activating the manager

Lesson objectives:
By the end of this lesson, it is anticipated that you will be familiar with the Script Manager basics.

### Lesson pre-requisites

You have to be familiar with the DSS User Interface basics to take this lesson.

### The DSS Script Manager components

Figure 1~~Figure 2~~ shows the components of the DSS Script Manager, namely:

1. The Scripts Explorer: where scripts are organized in user defined groups and subgroups or by storage (i.e. files).
2. The scripts view: where scripts are created, modified, debugged for errors and saved.
3. Tools Explorer: in this case it is used only to export and import scripts definitions. Since it is not used much for scripts it is not further described in this module.
4. The Properties window: where the selected script data is displayed.

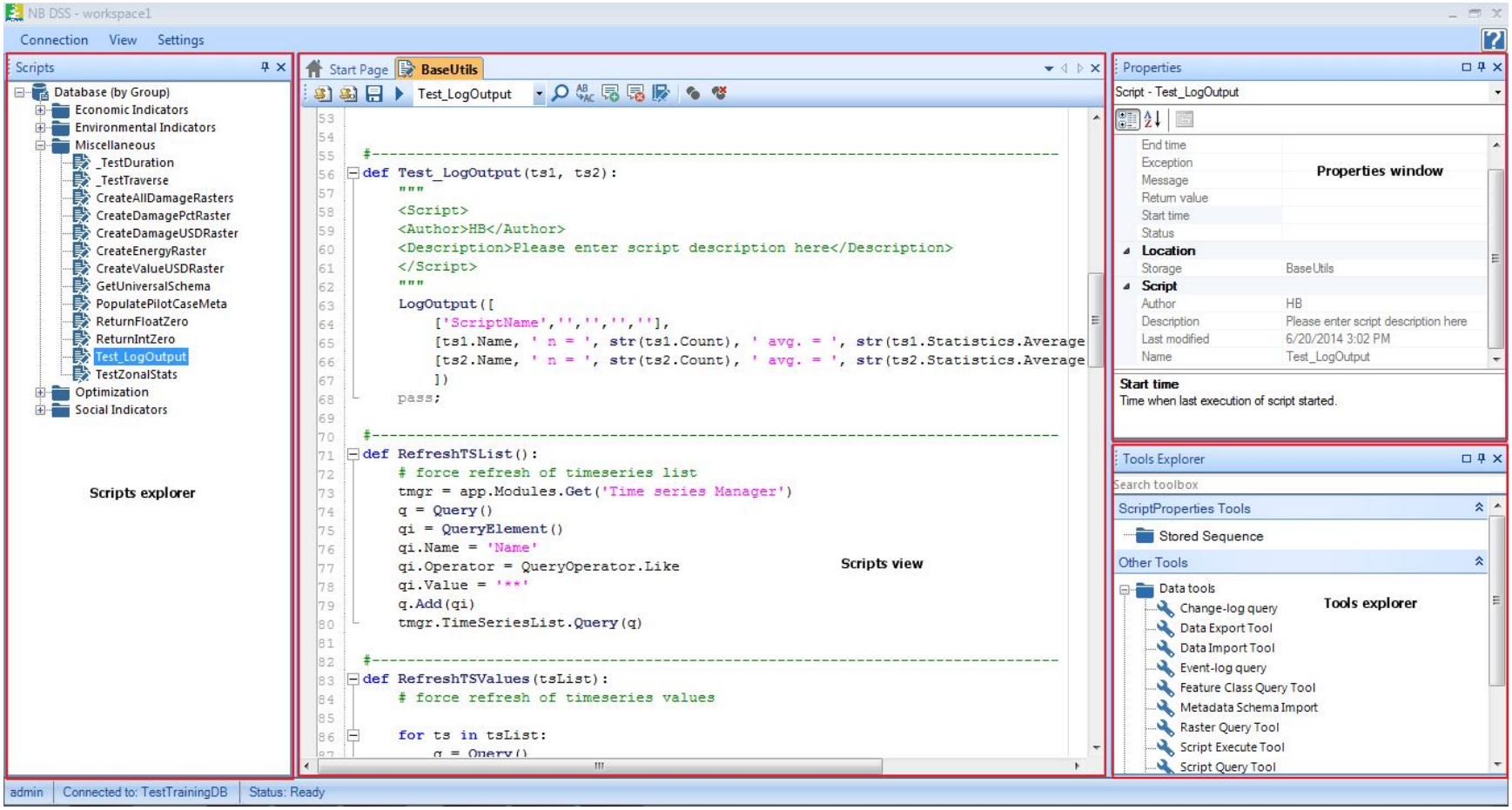



**Figure 1: Script Manager components**

## How scripts are stored in the DSS

When a new database is created, the Scripts Explorer window has only one main group which is the Database as shown in Figure 2~~Figure 3~~. Next to the 'Database' node, you can see that between parentheses 'by Group' is written.

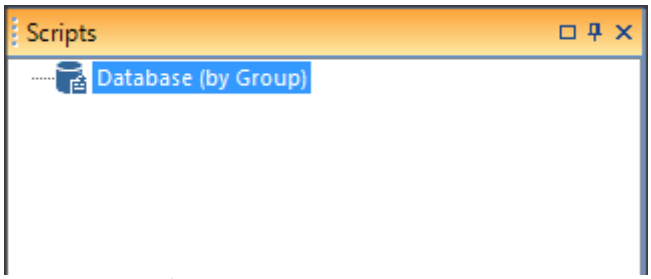



**Figure 2: Scripts explorer for newly created databases (showing database by group)**

Right click the database and select view by 'Storage' as shown in Figure 3.

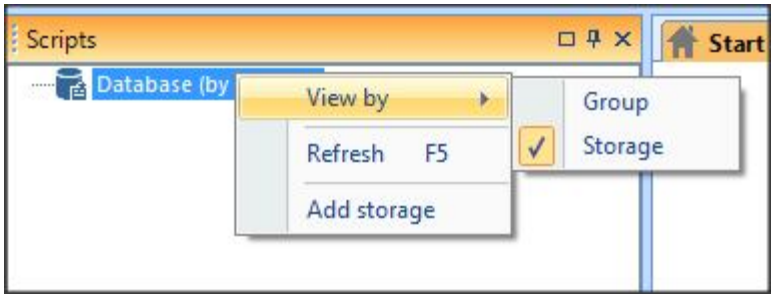



**Figure 3**

Now the text next to the 'Database' node changes to 'by Storage' as shown in Figure 4.

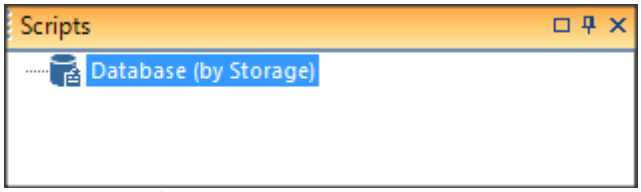



**Figure 4: Scripts explorer for newly created databases (showing database by storage)**

Therefore as you probably expected, Scripts can be viewed either by group or storage. Viewing by group is similar to arranging scripts in folders to easily access them (similar to other DSS objects in other explorers). So what is viewing by storage? A storage is equivalent to a file. In the DSS, a storage contains one or more scripts and functions coded in IronPython. Before you can add scripts to the DSS, you need first to add a new

storage. This is done by first viewing the scripts in 'by Storage' view, then right click the 'Database' node and select 'Add storage' as shown in Figure 5.

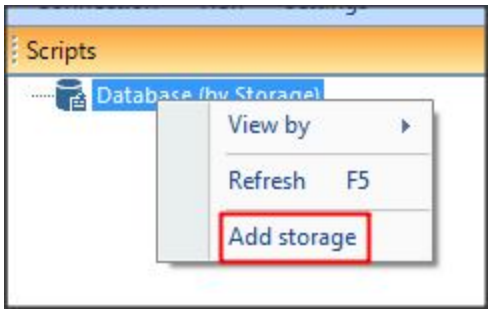



**Figure 5: Adding a storage to the database**

This adds a storage as shown in Figure 6.

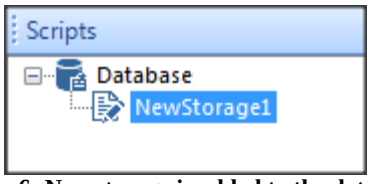



**Figure 6: New storage is added to the database**

## Script types in the DSS

The scripts in the DSS have the following two types:

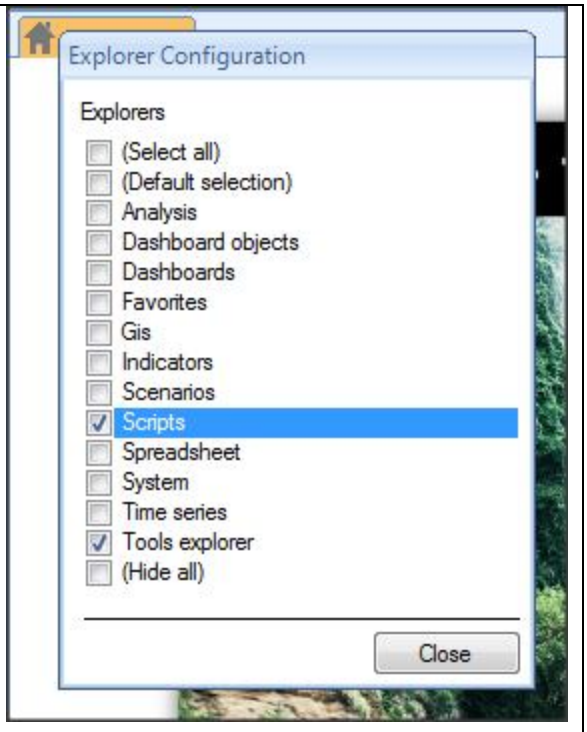
- Scripts with no arguments (i.e. simple scripts) which does not need arguments (i.e. data) to be passed to it before running. So it is a self-contained script that has all the data and code that are needed to run.
- Scripts with arguments (i.e. complex scripts) which does need arguments (i.e. data) to be passed to it before running. So it is a self-contained in terms of code but not data.

In addition, there are functions, which can only be called internally by other functions of scripts but are hidden when indicators are defined. Scripts are differentiated from functions by the header which is only required for scripts. For scripts to be used to calculate indicators, they have to return a single numeric output.

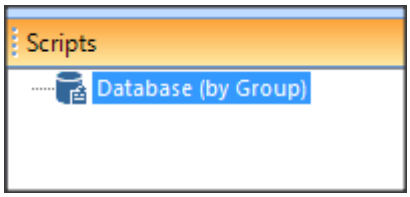
## Exercises

### *Activating the script Manager*

1- In the DSS, click on View Menu, click "Explorers…" and the Explorer Configuration box appears. Tick the box next to 'Scripts' explorer.

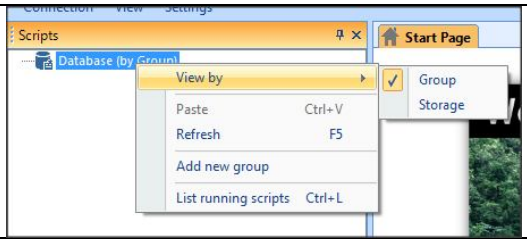


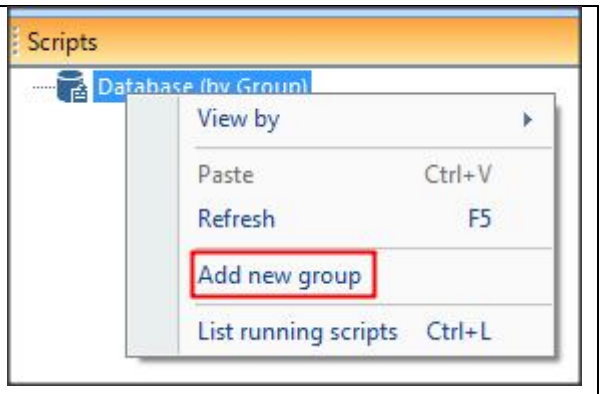
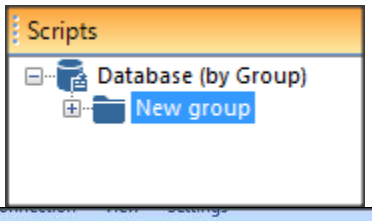
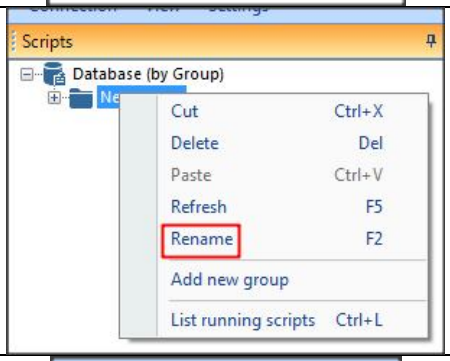
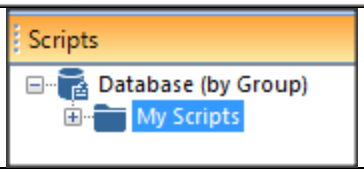

'Scripts' explorer should appear within the DSS window. The explorer has a root node 'Database'.




### *Adding a new 'user defined' group*

1- In the 'scripts' explorer, ensure that the scripts are shown by group as show next.



| | |
|---|---|
| 2- Right click on the 'Database' group, click on the Add new group option.<br><br>A new group is added as shown next. |  <br>  |
| 3- Select the new group and either right click with the mouse and select Rename or press the keyboard function Key 'F2' to rename it. |  |
| 4- Enter a suitable name (e.g. My Scripts). |  |

## Review Questions

1. List the components of the Script Manager.
2. Scripts can be viewed by group or storage in the DSS – explain the difference.
   - True
   - False

## Answers

1. Indicator Manager components are:
   - The Scripts Explorer
   - The scripts view
   - Tools Explorer window.
   - The Properties window.
2. True. Storages are similar to files which can contain several scripts and/or functions while groups are a visual grouping of scripts (only) in a tree like structure.

## 2.4. Creating simple scripts

### Introduction

This lesson shows you how you can add a new simple script.

Topics covered in this lesson:

- Create a simple script
- Debug a simple script
- Save a simple script

Lesson objectives:
By the end of this lesson, it is anticipated that you will be familiar with the process of creating simple scripts in the DSS.

### Lesson pre-requisites

You have to be familiar with scripts' basics and Iron Python (See the scripts' basics and the IronPython primer sections for details) to take this lesson.

### Script details

To make a script known to the DSS (i.e. its name appears within the explorer when view by group), it must have a header defining its author, and description for simple scripts and input (if with arguments) and output (if it returns a value) for complex scripts. The header for a simple script is shown in Figure 7.

```
def ScriptName():
    """
    <Script>
    <Author>admin</Author>
    <Description>Please enter script description here</Description>
    </Script>
    """
    # write your code here
    pass;
```

Script header

**Figure 7: Script header**

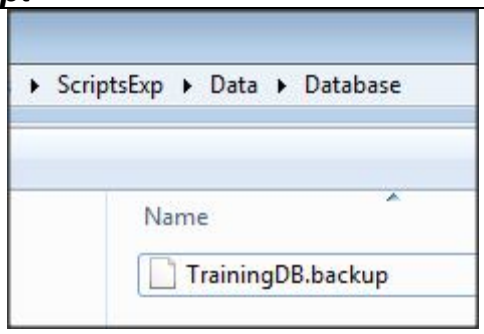
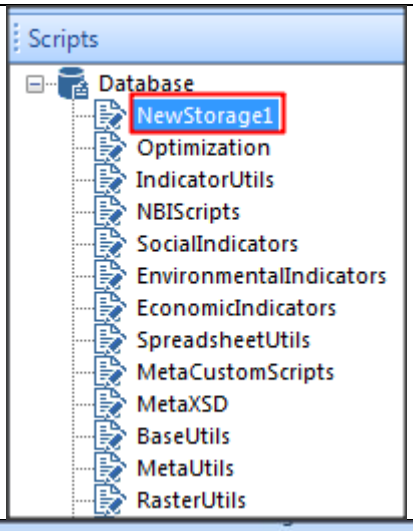
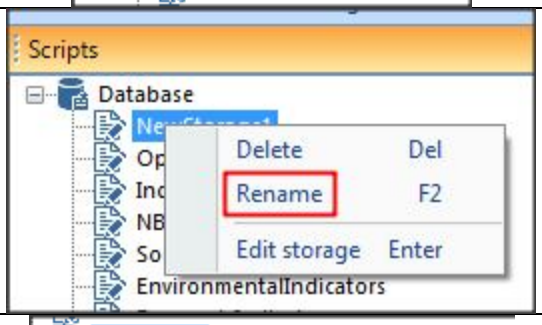
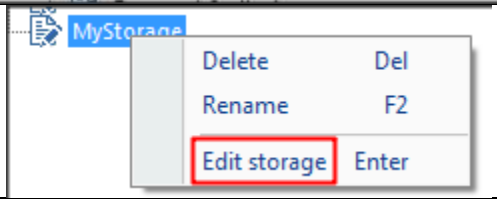
Scripts that are added without such headers, they become only local to the storage where they are saved and cannot be called directly from the DSS explorer. Scripts within one storage can call each other even if they have no headers.

## Script debugging

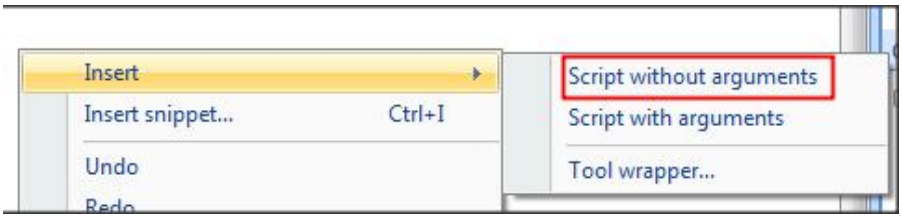
Script debugging can be helpful in understanding the execution of scripts as you can run the script line by line and/or stop execution at selected locations (breakpoints). This can also help to identify code flaws and errors. The DSS has got its own script debugger which allows you to debug a script code.

## Exercises

### *Adding, saving and running a simple script*

| | |
|---|---|
| 1- Restore the 'TrainingDB' database located in the folder **..\ScriptsExp\Data\Database**. Then create a connection and login to the restored database (For details see the Database Manager Utility and System Manager training module). Following that, activate and view the scripts explorer 'by storage' and a new storage as shown in the Activating the script Manager and How scripts are stored in the DSS sections. |  |
| 2- In the restored database, you will notice that there are already created storages in addition to the newly created storages that is called 'NewStorage1'. |  |
| 2- Right click the newly created storage and select rename to rename to 'My Storage'. |  |
| 3- If the storage is not already opened in the scripts view, right click it and select 'Edit storage'. |  |

4- In the script view, right-click and select 'Insert' then 'Script without arguments' (Note that you can alternatively do this by clicking  the button on the toolbar).




This will insert template code (see below) for a simple script including a header. The script does nothing.

```
def ScriptName():
    """
    <Script>
    <Author>admin</Author>
    <Description>Please enter script description here</Description>
    </Script>
    """
    # write your code here
    pass;
```

5- Rename the script and then modify it to simply write the word “Hello” to the console.

to do this:

1- change the function name from 'ScriptName' to 'MyFirstScript'

2- change the line:

pass;

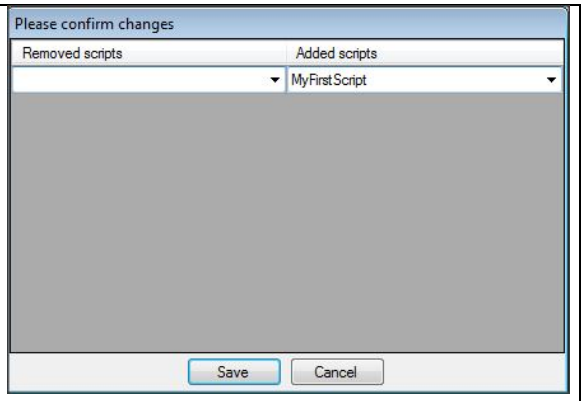
to

print 'Hello';

Script should look like the window below

```
def MyFirstScript():
    """
    <Script>
    <Author>admin</Author>
    <Description>Please enter script description here</Description>
    </Script>
    """
    # write your code here
    print 'Hello';
```

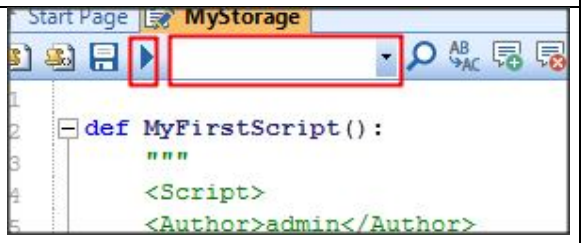
6- Save the storage by clicking the  on the toolbar. A confirmation window appears, showing what scripts are added and which are removed (if any). Obviously in this case only one script is added and nothing was removed. Click 'Save' to confirm. This will create the 'MyFirstScript' as something the DSS knows (How can you check that? Hint: Change the view to 'by Group').




7- Now it is time to run the script. In the toolbar, there is an arrow followed by a list box as shown next.

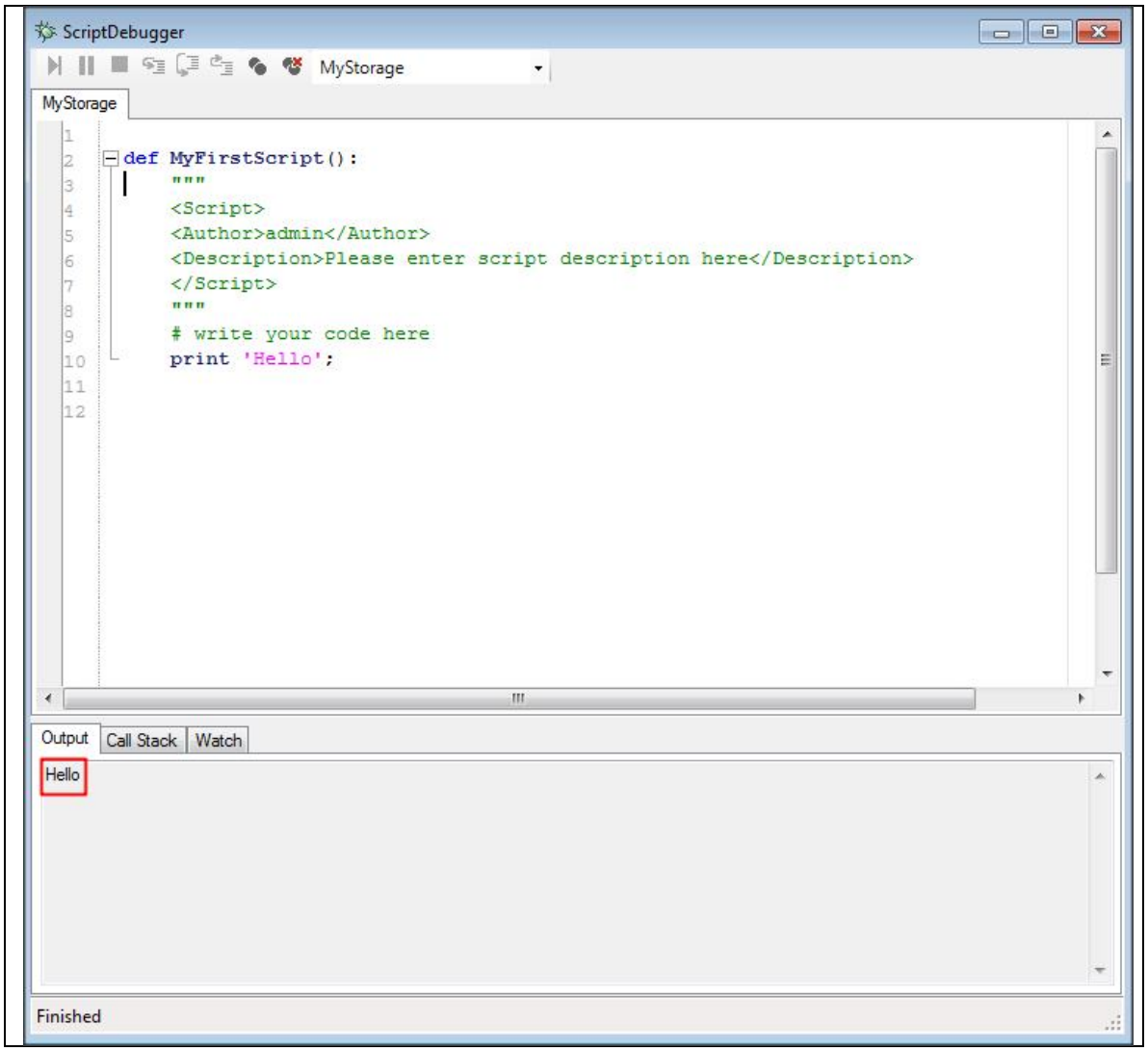
The list box allows you to select the script that you need to run and the  button allows you to run this script.

So select the 'MyFirstScript' in the box and then click the  button.




8- The script debugger windows appears and the 'Hello' word appears in the 'Output window' as shown below.

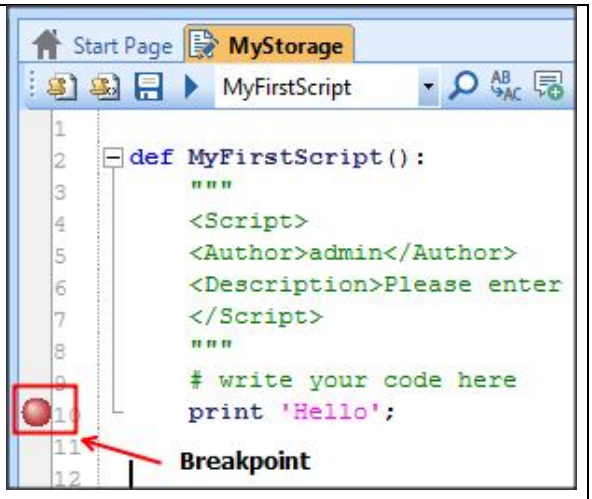
The script debugger has got the code in the top pane and the script output in the lower output pane. The lower pane has also two other tabs that are called 'Watch' and Call stack'. The 'Watch' windows allows you to watch the value of the script variables when running in step by step or using break points. The 'Call stack' windows allows you to see the current execution point of the the script and a list of functions and scripts called at the point of execution.


ScriptDebugger
MyStorage
MyStorage
def MyFirstScript():
"""
<Script>
<Author>admin</Author>
<Description>Please enter script description here</Description>
</Script>
"""
# write your code here
print 'Hello';
Output
Call Stack
Watch
Hello
Finished

### *Debugging a simple script*

1- To debug a script code, you need to establish stop location (i.e. breakpoints). You can do this by either:

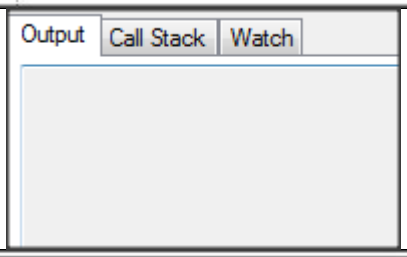
- Clicking the left border of the script view. This makes a breakpoint at this line which will stop execution here.
- Moving the cursor to a code line 10 and pressing F9. This will also establish a breakpoint.



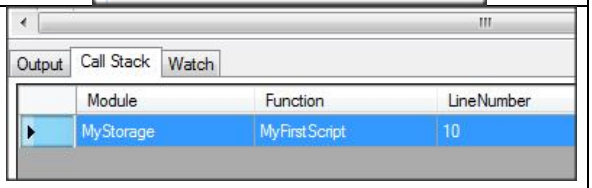

Breakpoints can be removed/toggled in the same way they were created. The  buttons in the toolbar allows you to delete all breakpoints or disable them during execution respectively.

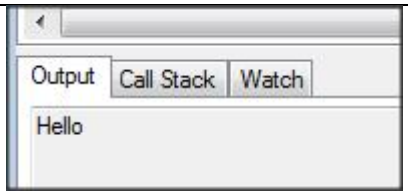
2- Run the script as done in the previous exercise. Now the script debuger appears but it stops at the breakpoint line as indicated by the yellow color shown next.


3- Now look at the 'Output' window of the debugger. What do you notice?

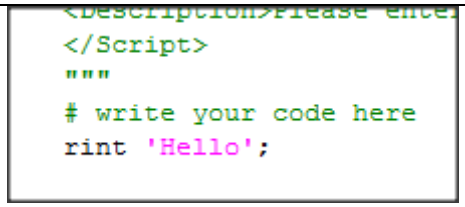
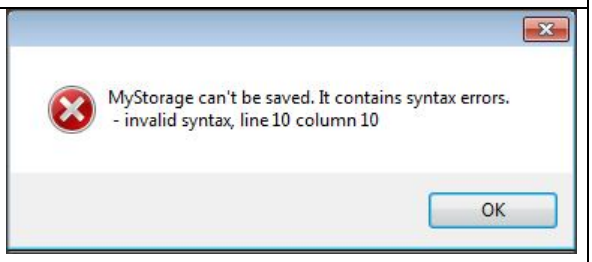
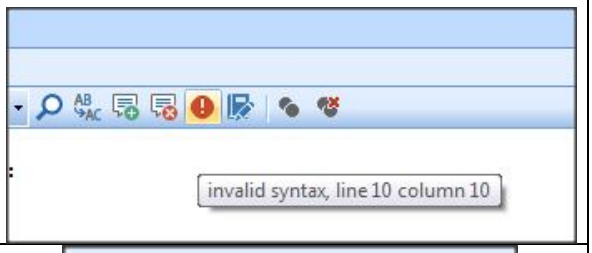



4- Move the the 'Call Stack' window of the debugger. It shows the current execution point which is line 10 in 'MyFirstScript' in storage 'MyStorage'. Does that explain why the 'Output' window of the debugger



| | |
|---|---|
| is empty. | |
| 5- Go back to the 'Output' tab and click the  button twice on the debugger toolbar to contniue running the script. Now the output window shows the word 'Hello' |  |

***Identifying script errors***

| | |
|---|---|
| 1- In 'MyFirstScript', change the line:<br>print 'Hello'<br>To<br>rint 'Hello'<br>then try to save or run the script. What do you notice? |  |
| 2- An error message appears warning you that the storage cannot be save as there are some syntax errors. It also shows you in which line and column the error is. Click Ok. |  |
| 3- Note also the red exclamation mark that is added to the toolbar.<br><br>Hover the mouse over it and the error message appears. |  <br>  |
| 4- Correct the error by changing the line:<br>rint 'Hello'<br>To<br>print 'Hello'<br>Save the script.<br>Note that the red exclamation mark disappeared. |  |

## Review Questions

1- What are the details that are needed for a complex script to be known to the DSS?

2- Script debugging cannot be done within the DSS.

- True
- False

## Answers

1- To make a complex script known to the DSS, it must have a header defining its author, and description, input and output (if it returns a value).
2- False.

## 2.5. Handling changes and metadata

### Introduction

This lesson introduces you to the handling of script changes and metadata within the DSS.

Topics covered in this lesson:

- Examining the change log entries for a script
- Importing and editing a script metadata

Lesson objective:

After completing this lesson, you will be able to:

- Understand the change log entries for each script
- Handle script metadata

### Lesson pre-requisites

You have to be familiar with script manager basics (See the Script Manager basics section for details) to take this lesson.

### Script storage changes and metadata

One of the main challenges to data users is to keep a log of the changes made to a data set and also save and keep its metadata updated. The DSS solves this problem though an innovative solution. When a script storage is added to the Script Manager, The DSS monitors all operations that is carried out on it noting the time and date of this operation, and who carried it out. For example, when the storage is added, an entry is added to the 'Change log' of this it to show the time and date of adding this storage and also a description of the operation as shown in the below figure. Not that this applies only to a whole storage not to individual scripts/functions, therefore, you can see the change log and metadata tabs of properties when viewing by storage only.

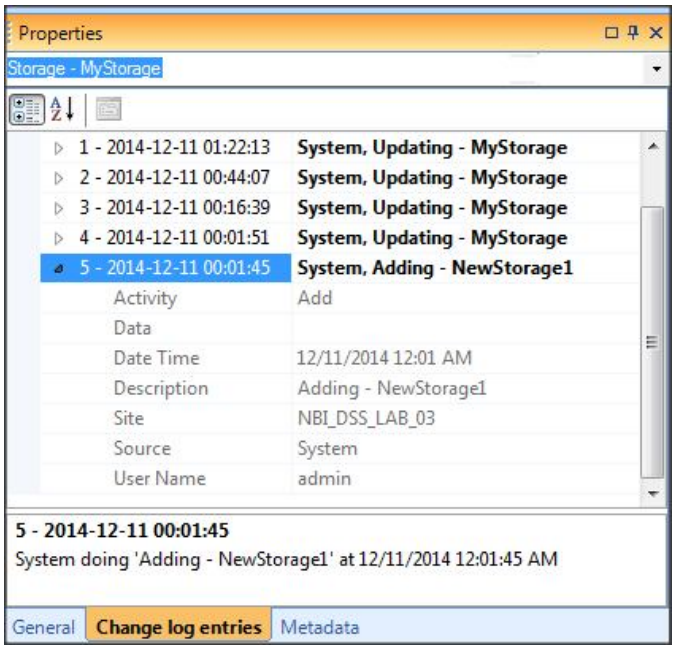



**Figure 8: Change log example**

Similarly, the DSS allows you to import storage metadata (if exists) through an xml schema. Once this schema is within the DSS, it is saved and linked to all storages where the metadata can be entered and updated as needed.

To define the metadata properties an agreement on a common set of metadata properties to be used has to be made. At a technical level the metadata properties must be expressed as an XML schema. An example of a simple schema is:

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="metadata" > <!--Root node -->
<xs:complexType>
<xs:sequence>
<xs:element name="identification" minOccurs="0" > <!--Category -->
<xs:complexType>
<xs:sequence>
<xs:element name="originator" type="xs:string" minOccurs="0" />
<xs:element name="publicationdate" type="xs:dateTime" minOccurs="0" />
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:sequence>
</xs:complexType>
</xs:element>
</xs:schema>
```

The above simple schema defines one property, `identification`, which is optional (i.e. minoccurs=0) and consists of two (also optional) values, `originator` and `publicationdate`. The first is a string, while the latter is a date-time.
Data types of properties in such a schema should be kept to standard types as defined by http://www.w3.org/2001/XMLSchema

A more elaborate sample is this – but still constructed following the line from above:

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema attributeFormDefault="unqualified" elementFormDefault="qualified"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="metadata" >
    <xs:complexType>
      <xs:sequence>
        <xs:element name="identification" minOccurs="0" >
          <xs:complexType>
            <xs:sequence>
              <xs:element name="originator" type="xs:string" minOccurs="0" />
              <xs:element name="publicationdate" type="xs:dateTime" minOccurs="0" />
              <xs:element name="description" type="xs:string" minOccurs="0" />
              <xs:element name="timeperiodofdata" minOccurs="0" >
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="fromdate" type="xs:dateTime" minOccurs="0" />
                    <xs:element name="todate" type="xs:dateTime" minOccurs="0" />
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
              <xs:element name="progress" type="xs:string" minOccurs="0" />
              <xs:element name="securityclassification" type="xs:string" minOccurs="0"
/>
              <xs:element name="securityhandlingdescription" type="xs:string"
minOccurs="0" />
              <xs:element name="contactperson" type="xs:decimal" minOccurs="0" />
              <xs:element name="contactorganization" type="xs:string" minOccurs="0" />
              <xs:element name="contactemail" type="xs:string" minOccurs="0" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="dataquality" minOccurs="0" >
          <xs:complexType>
            <xs:sequence>
              <xs:element name="logicalconsistencyreport" type="xs:string"
minOccurs="0" />
              <xs:element name="accuracyreport" type="xs:string" minOccurs="0" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="spatialreference" minOccurs="0" >
          <xs:complexType>
            <xs:sequence>
              <xs:element name="geographiccoordinatesystemname" type="xs:string"
minOccurs="0" />
              <xs:element name="latituderesolution" type="xs:decimal" minOccurs="0" />
              <xs:element name="longituderesolution" type="xs:decimal" minOccurs="0" />
```

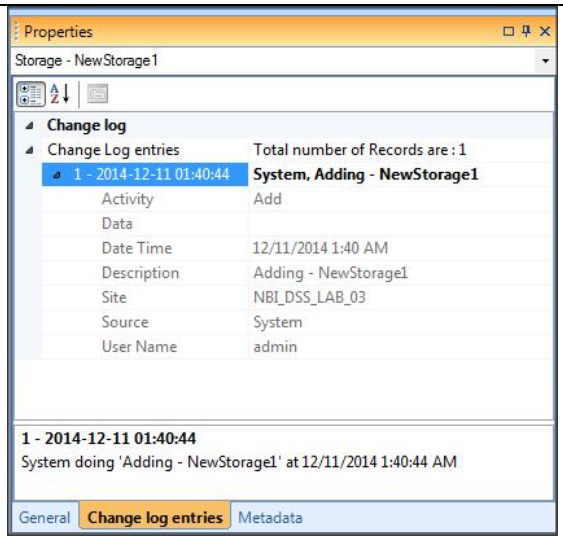
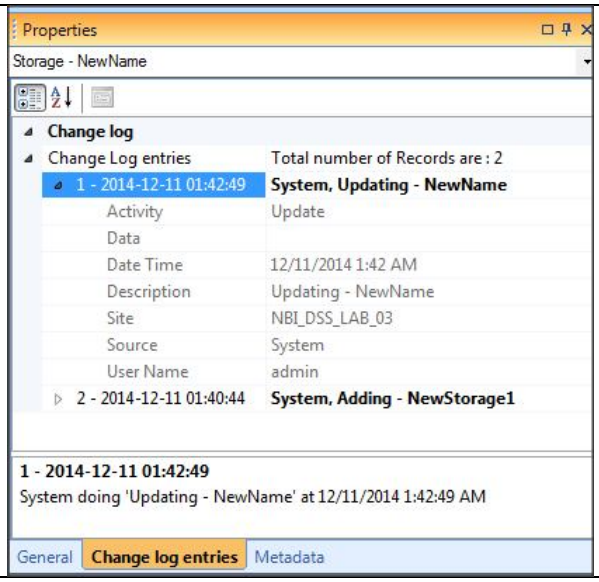
```
              <xs:element name="geographiccoordinateunits" type="xs:string"
minOccurs="0" />
              <xs:element name="unitofdatavalues" type="xs:decimal" minOccurs="0" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

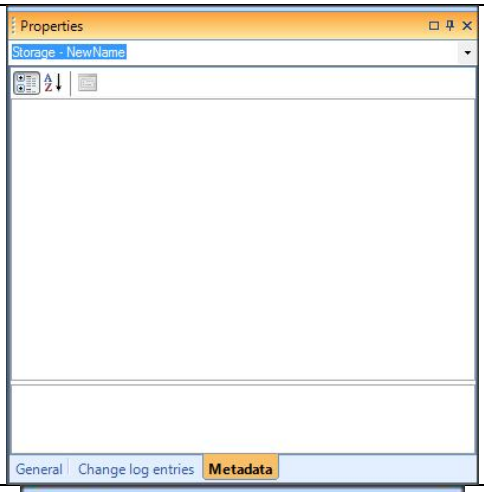
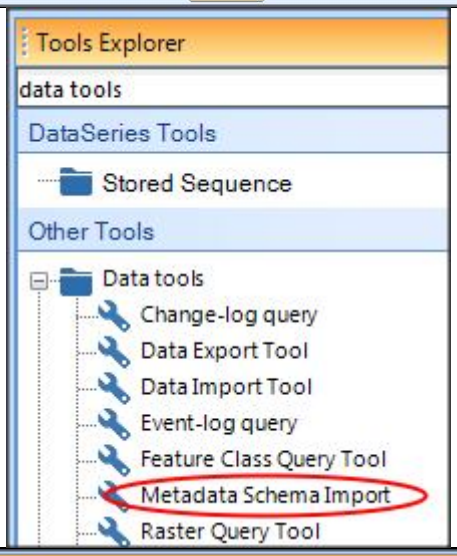
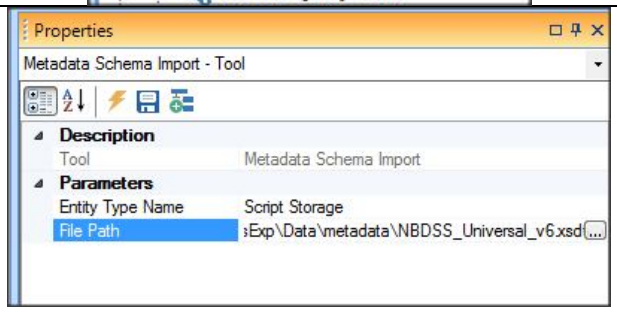
Note in addition to 'string' and 'datetime' data types, 'decimal' types are also used.

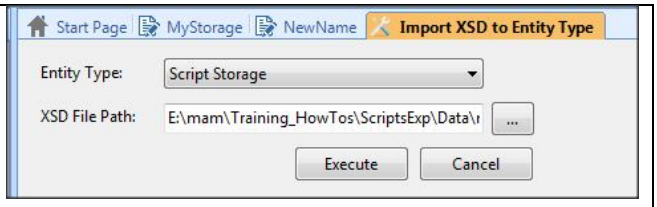
You are encouraged to read more about the xml schema in the 'Data Quality Assurance Guideline: Data Processing, Quality Assurance and Metadata' report that was published as part of the 'Data Compilation and Pilot Application of the Nile Basin Decision Support System (NB-DSS)' study (Work Package 2: Stage 2).

## Exercises

### *Handling time series change Log and metadata*

| | |
|---|---|
| 1- Add a storage into the Script Manager (See How scripts are stored in the DSS section for details). In the Properties Windows, Select the 'Change log entries' tab. You will notice that there is one entry in the change log. The entry shows that the storage was added to the database. Double click the entry to expand (or alternatively click the little arrow to the left of the entry). You can see more details such as the activity type, date and time and user who carried out the activity. |  |
| 2- Rename the storage and check the again the 'Change log entries' tab.<br><br>What did you notice? Write down your observations. (Hint: compare what you see against the next figure).<br><br>Please also not how the entries are ordered. |  |

| | |
|---|---|
| 3- To handle metadata, a third tab also exists for script storages which stores its metadata. For the existing storages no metadata fields exists. |  |
| 4- If no metadata fields does not exist, it can be imported using the 'Metadata Schema Import' tool under the 'Data tools' category. To use the tool, select 'Metadata Schema Import' from the 'Data tools' category. |  |
| 5- Once the tool is selected, its properties appear in the 'Properties' window. Two parameters need to be entered for this tool. The first is the DSS entity type (i.e. script storage in this case') and the second is the 'File Path' to the metadata schema file. Select the 'NBDSS_Universal_v6.xsd' file that is located in the '**..\ScriptsExp\Data\metadata**' folder. |  |

| | |
|---|---|
| 6- Click the  button. The next dialog box appears. Confirm that both entity type and XSD file path are correct and then click the  button. |   |
| 7- The metadata schema is imported and loaded into the 'Meta data' tab. Familiarize yourself with the content. You may start adding data to the different metadata fields. | |

## Review Questions

1. Explain how storage metadata schema is imported and maintained with the DSS.
2. The DSS keeps track of all the operations made on a script.
   - True
   - False
3. When a script storage metadata schema is imported into the DSS, can this schema be made available for time series data?
   - True
   - False

## Answers

1. The DSS allows the users to import script storage metadata schema through an xml schema using the ‘Metadata Schema Import’ tool. Once this schema is within the DSS, it is saved and linked to the storage. Metadata can also be updated directly by the users if needed.
2. False.
3. False. A specific entity type is specified for each metadata schema at the time it is imported into the DSS database.

## 2.6. Creating complex scripts

### Introduction

This lesson shows you how you can add a new complex script.

Topics covered in this lesson:

- Create a complex script
- Debug a complex script
- Save a complex script

Lesson objectives:
By the end of this lesson, it is anticipated that you will be familiar with the process of creating complex scripts in the DSS.

### Lesson pre-requisites

You have to be familiar with scripts' basics, simple scripts and Iron Python (See the scripts' basics, simple scripts and IronPython primer sections for details) to take this lesson.

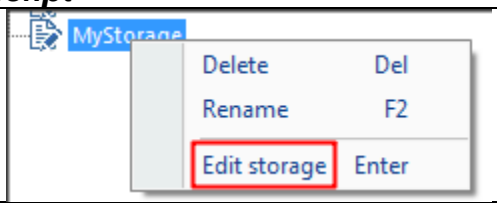
### Script arguments

The complex scripts are different from the simple ones as they have arguments. They have to receive those arguments first before they run. They can receive them directly, or from other simple or complex scripts. These arguments can be numbers, text or even a DDS object (e.g. a time series or a scenario). In this section, creating a complex script that takes two numbers as arguments will be described. In the advanced scripting section, creating a complex script with a DSS object argument will be presented.

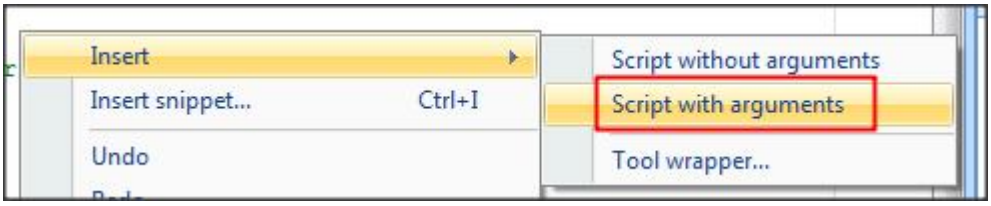
## Exercises

### *Adding, saving and running a complex script*

1-Select 'My Storage', right click it and select 'Edit storage'.




2- In the script view, right-click and select 'Insert' then 'Script with arguments' (Note that you can alternatively do this by clicking  the button on the toolbar).




This will insert template code (see below) for a simple script including a header. The script does nothing.

```
def ScriptName(param1, param2):
    """
    <Script>
    <Author>admin</Author>
    <Description>Please enter script description here</Description>
    <Parameters>
    <Parameter name="param1" type="int">Parameter of type int</Parameter>
    <Parameter name="param2" type="IType">Parameter of type IType</Parameter>
    </Parameters>
    <ReturnValue type="IType">Function returns object of type IType</ReturnValue>
    </Script>
    """
    # write your code here
    pass;
```

3- Rename the script and then modify it to simply sum two numbers. to do this:

- change the function name from 'ScriptName' to 'SumTwoNumbers'
- change the script body to the following:

```
"""
    <Script>
    <Author>admin</Author>
    <Description>Please enter script description here</Description>
    <Parameters>
    <Parameter name="param1" type="int">Parameter of type
int</Parameter>
```

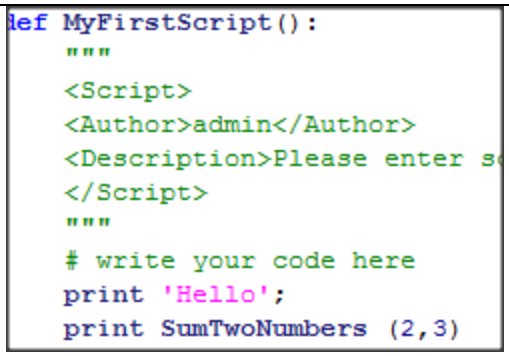
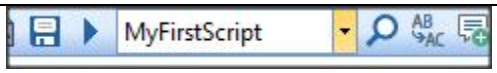
```
    <Parameter name="param2" type="double">Parameter of type
double</Parameter>
    </Parameters>
    <ReturnValue type="double">Function returns object of type
double</ReturnValue>
    </Script>
    """
    # write your code here
    return param1 + param2
    pass;
```

Note the changes that were made to the arguments (i.e. parameters).

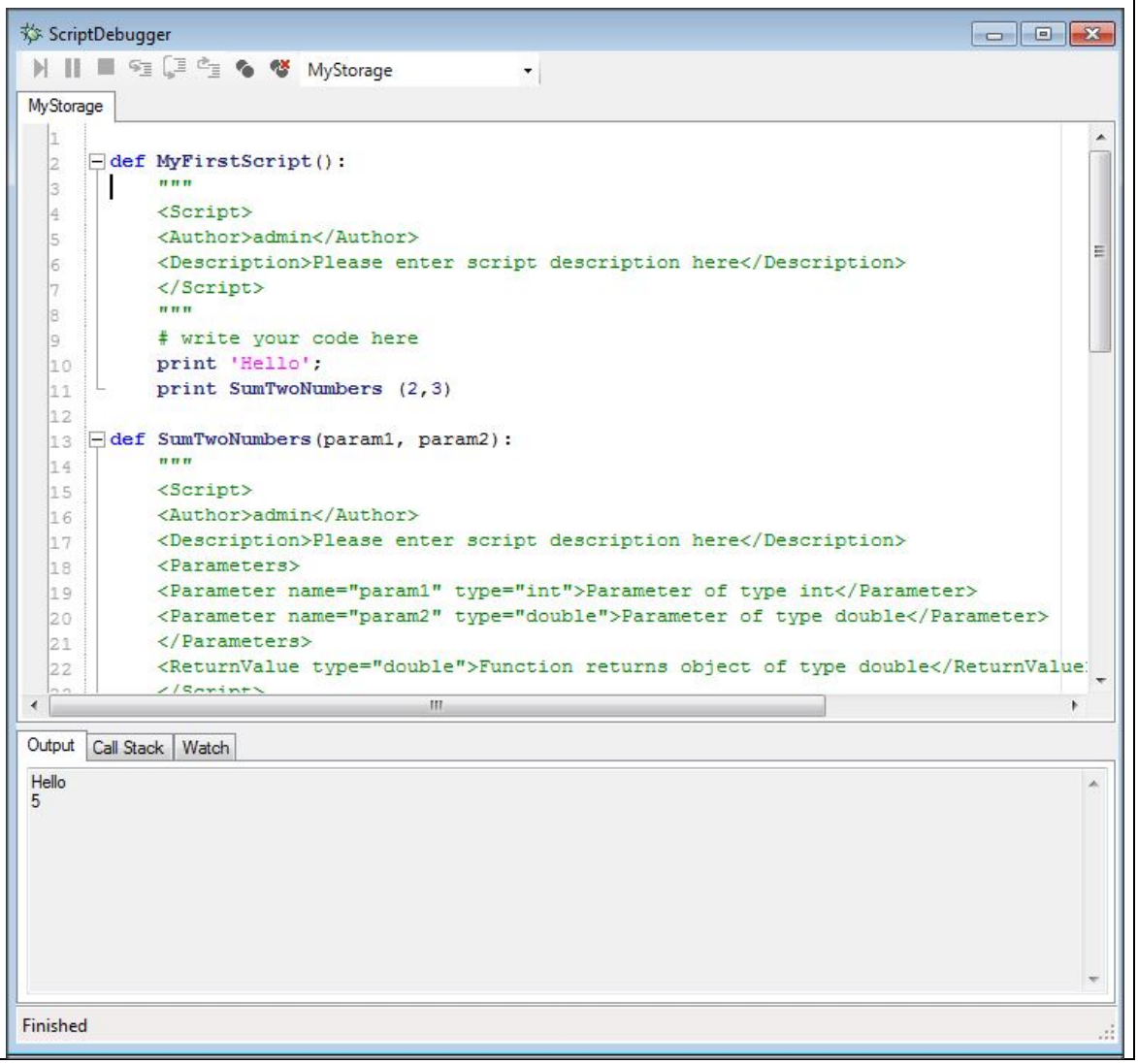
The script should look like the window below

```
def SumTwoNumbers(param1, param2):
    """
    <Script>
    <Author>admin</Author>
    <Description>Please enter script description here</Description>
    <Parameters>
    <Parameter name="param1" type="int">Parameter of type int</Parameter>
    <Parameter name="param2" type="double">Parameter of type double</Parameter>
    </Parameters>
    <ReturnValue type="double">Function returns object of type double</ReturnValue>
    </Script>
    """
    # write your code here
    return param1 + param2
    pass;
```

Save the storage by clicking the  on the toolbar and click save to confirm adding the complex script.

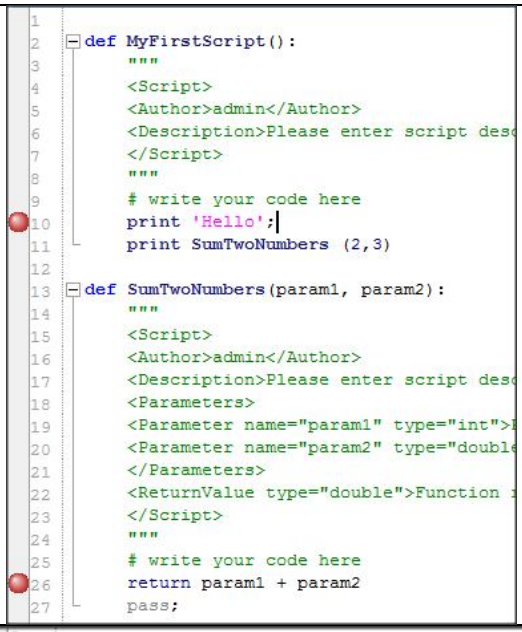
| | |
|---|---|
| 4- Now modify the 'MyFirstScript' script to call the 'SumTwoNumbers' script as follows:<br>Add the following line<br>print SumTwoNumbers (2,3) |  |
| 5- Run the 'MyFirstScript' script by selecting it the 'MyFirstScript' in the box and then click the  button. |  |

The script debugger windows appears and the 'Hello' word appears in the 'Output window' as shown below followed by the sum of 2 and 3 (i.e. 5).



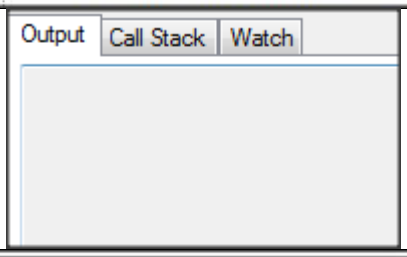
### *Debugging a complex script*

1- Add two breakpoints as shown next. The first is within the 'MyFirstScript' script at the first print line. The second in within the 'SumTwoNumbers' script at the return line.

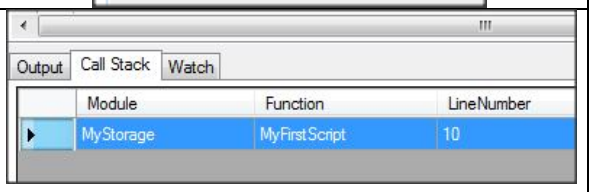



2- Run the 'MyFirstScript' script as done in the previously. Now the script debuger appears but it stops at the breakpoint line as indicated by the yellow color shown next.


3- Now look at the 'Output' window of the debugger. What do you notice?

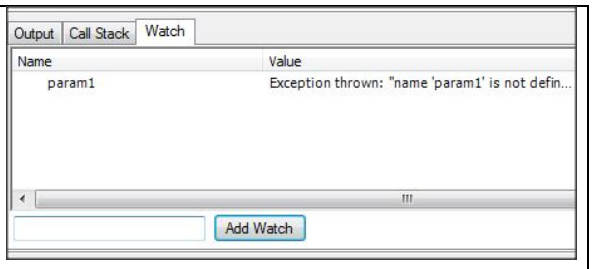



4- Move the the 'Call Stack' window of the debugger. It shows the current execution point which is line 10 in 'MyFirstScript' in storage 'MyStorage'.

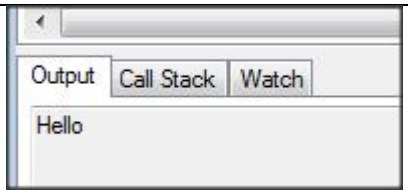


5- Now move to the 'Watch' window and a add watch to inspect a variable. To do this, write 'param1' in the input box and click 'Add watch'. A line is added in the list.

Note that the Watch window shows an exception because param1 is not known in the 'MyFirstScript' and cannot therefore be evaluated.

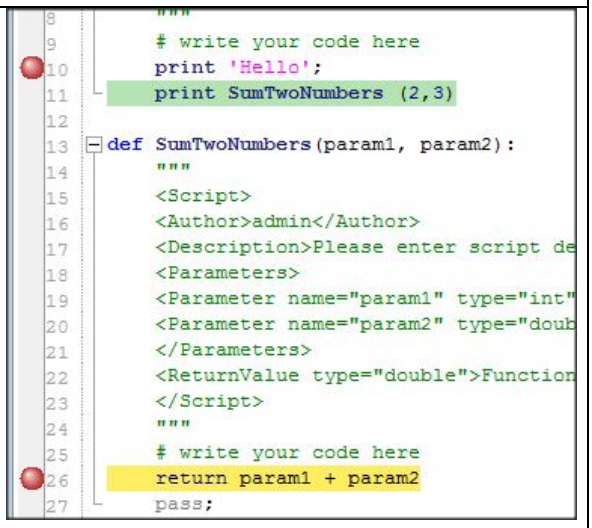



5- Go back to the 'Output' window and click the  button on the debugger toolbar (or press F10) to execute the current line and move forward to next line. Now the output window shows the word 'Hello'

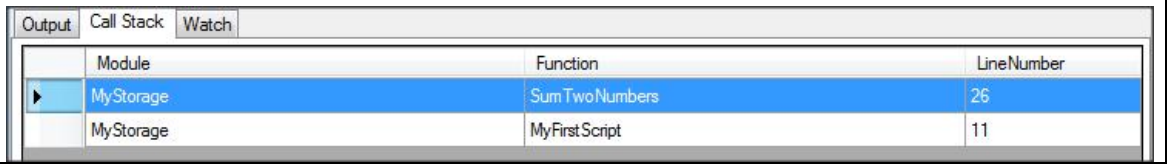



6- Now click the  on the debugger toolbar (or press F5) to continue the execution until the next breakpoint.

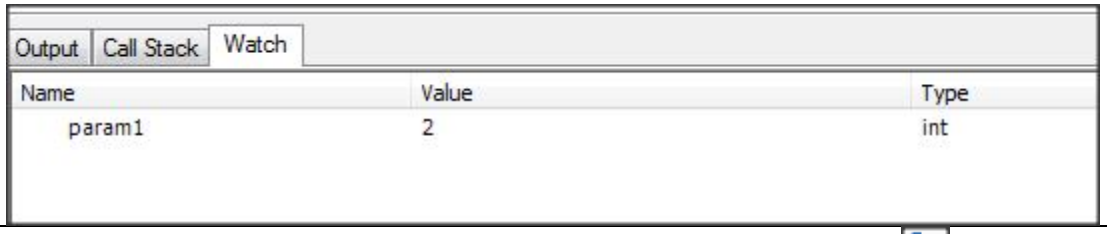
Note that the following:
- The yellow color moved to the return line in the 'SumTwoNumbers' script indicating the current statement.
- The green color indicating the active statement in 'MyFirstScript' calling 'SumTwoNumbers'



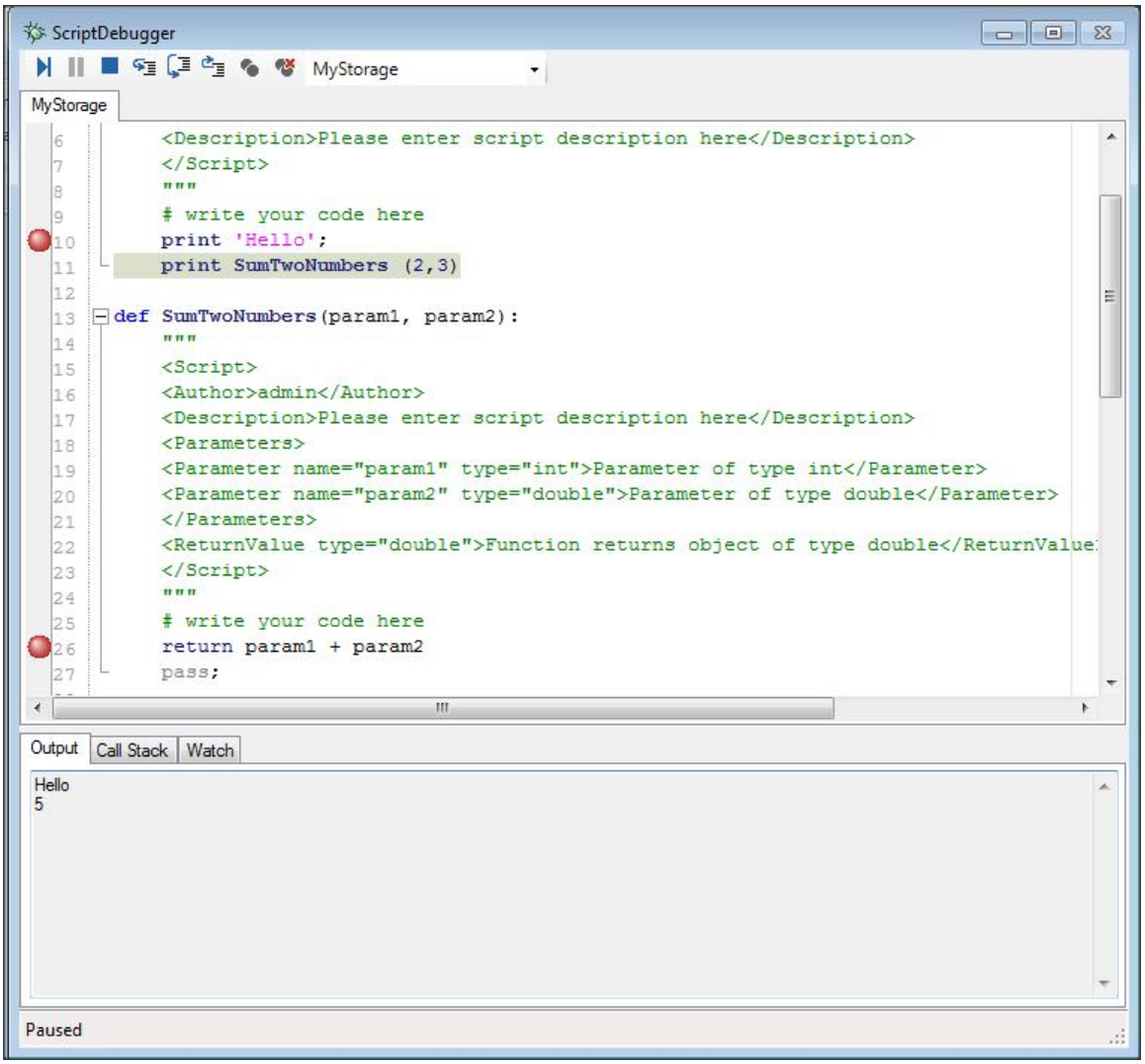

7- Now move to the 'Call Stack' window. Where is the currect excution point? (Hint: the line at the top of the call stack shows the current execution point)



8- Now move to the 'Watch' window. Since we are within the 'SumTwoNumbers' Script, the debugger was able to show the value of param1 which was passed from the 'MyFirstScript' (i.e. 2).




9- Now move back to the 'Output' windows and window and click the  button on the debugger toolbar (or press F10) twice. Now the sum of the two numbers is printed and the execution is back to 'MyFirstScript'




Check the 'Call stack' and 'Watch' windows and writedown your observations then click

| |
|---|
| the  button to finish the sript run. |

## Review Questions

1. Give three examples of complex script arguments?
2. DSS objects cannot be passed to a complex script.
    - True
    - False

## Answers

1. Complex script arguments can be:
    - Numbers
    - Text
    - DSS objects
2. False (they can be passed).

## 2.7. Predefined scripts in the DSS

### Introduction

This lesson gives an overview of the DSS predefined scripts. It also shows you how you can expand those predefined scripts.

Topics covered in this lesson:

- Who developed this set of predefined scripts
- Definition of each script showing its function.
- Expanding the predefined scripts in the DSS.

Lesson objective:

After completing this lesson, you will be familiar with the predefined scripts in the DSS and you will know how to expand those scripts

### Lesson pre-requisites

You have to be familiar with scripts basics (See the indicators' basics section for details) to take this lesson.

### Who developed this set of predefined scripts

During the development of the Nile basin DSS, a number of consultation meetings and workshops were held to identify the key indicators that stakeholders in the Nile Basin are most interested in. This was part of a consultancy called 'Data Compilation and Pilot Application of the Nile Basin Decision Support System'. Based upon the discussions between the stakeholders, the consultant identified a number of key indicators that can be used in the DSS to evaluate scenarios and undertake MCA and CBA. These indicators were divided into the following three categories:

- Social indicators
- Environmental indicators
- Economic indicators

A scripting library was developed for the calculation of the above indicators. The scripts are organized into the following eight storages:

- BaseUtils: Generic scripts for common mathematical calculations, interpolation, lookups, etc.
- SpreadsheetUtils: Scripts for accessing DSS spreadsheets and retrieving arrays and/or lookup values from the spreadsheets associated with the developed indicators.
- IndicatorUtils: Supporting scripts for calculating environmental, social and economic indicators and calculation of ecologically relevant time series statistics.
- NBIScripts: Scripts for calculation of food production indicators (Developed by NBI)
- RasterUtils: Scripts for raster processing, mainly for flood damage calculations.
- Environmental Indicators: Scripts for calculation of environmental indicators.
- Social Indicators: Scripts for calculation of social indicators.
- Economic Indicators Scripts for calculation of economic indicators.

Figure 9 shows the dependencies between the above script storages.

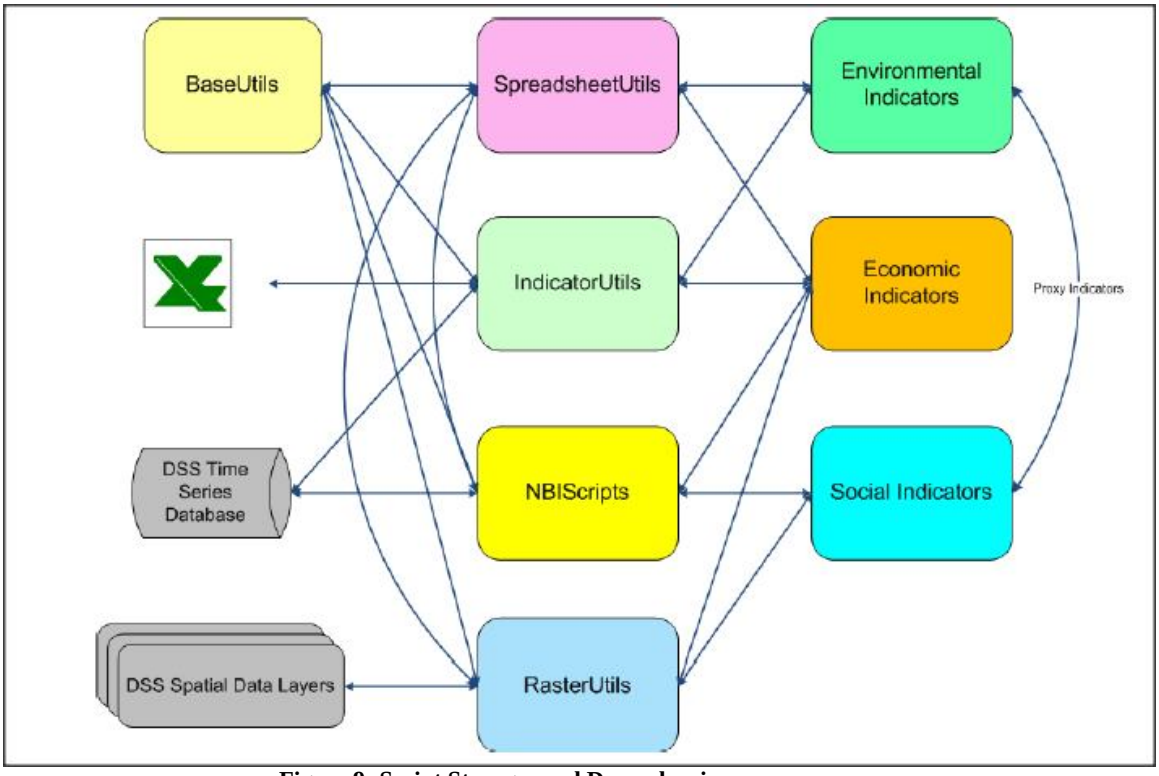



**Figure 9: Script Storages and Dependencies**

In the following section, definition of the scripts that are used to calculate the DDS indicators is given. For a full reference to the scripts within the above storages, you are referred to the ???? report. The storages are self-documented using comments.

## Scripts Definition

In this section, for each predefined script, the following is presented:

1. script sub-category
2. script name
3. a description of what the script does

### *Social Indicators*

| Sub-category | NB-DSS Script name | Description |
|---|---|---|
| Water Availability | SO1_WaterAvailability | Calculates the change in availability of water for riparian users: domestic consumption, subsistence agriculture and livestock |
| Community Health and Safety | SO2_MalariaEndemicity | Calculates the susceptibility of irrigation scheme areas to malaria based on WHO malaria incidence map for Africa |
| | SO3_PestDiseasePrevalence | Calculates the prevalence of diseases resulting from pest species |
| | SO4_UrbanPollution | Calculates the water pollution downstream major urban areas |
| | SO5_HouseholdsFlooded | Calculates the No households within the 100 year flood line |
| | SO6_DrowningRisk | Calculates the drowning risk due to conveyance of water in an open canal |
| Food security and Livelihoods | SO7_FormalIrrigation | Calculates the footprint area due to establishment of new irrigation schemes |
| | SO81_RecessionAgricFloodPlain | Calculates the impact on Recession agriculture due to floodplain inundation |
| | SO82_RecessionAgricBank | Calculates the impact on Recession agriculture due bank instability |
| | SO91_FishProductionDam | Calculates the change in fish productivity in a dam, lake, or wetland |

| Sub-category | NB-DSS Script name | Description |
|---|---|---|
| | SO92_FishProductionRiver | Calculates the change in fish productivity along a river reach |
| | SO10_ProductiveLandUse | Calculates the productive land use for crops, grazing inundated by dam or lost due to establishment of an irrigation scheme or a canal |
| | SO11_LossNaturalResources | Calculates the change loss of access to natural resources due to inundation by dam or establishment of an irrigation scheme or a canal |
| Displacement | SO12_PhysicalDisplacement | Calculates the physical displacement of population due to inundation by a dam, establishment of an irrigation scheme or construction of a canal |
| | SO13_EconomicDisplacement | Calculates the economic displacement due to disruption of access to natural resources (cattle, people, wildlife) as a result of a canal and/or a dam construction |

***Environmental Indicators***

| Sub-Category | NB-DSS Script Name | Description |
|---|---|---|
| Footprint Areas | EN1_EnvSensitiveAreas | Calculates the extent of Environmentally Sensitive Area within a dam, irrigation scheme or canal footprint |
| | EN11_EnvSensitiveRating | Calculates the impact rating on environmentally sensitive area within a dam, irrigation scheme or canal footprint |
| | EN12_HotspotRating | Determines the wetlands of international importance (Ramsar Sites) and Important Bird Areas (IBAs) that fall outside of protected areas, but within primary impact zones. |
| | EN2_Carbon | Calculates the area of woody biomass and biomass carbon within dam footprint |
| | EN3_FishProduction | Estimates fish production from a dam, lake or a wetland |
| Downstream Areas | EN4_FloodPlainInundation | Calculates the floodplain area inundated compared to a baseline |
| | EN42_WetlandArea | Calculates the wetland area inundated compared to a baseline |
| | EN5_EcoStressRating | Determines ecological stress rating from changes in key flow components and flow variability compared to baseline. |
| | EN6_WetDuration | Calculates the wet season duration based on median monthly flows |
| | EN7_BlackFlyRating | Determines black fly rating from HP operation, changes in low flows and variability compared to baseline. |
| | EN8_BankStability | Calculates bank stability rating downstream of impoundment based on standard deviation of flows and predefined sinuosity |
| | EN9_RecoveryDistance | Estimates recovery distance based on median discharge from impoundment and distance to downstream tributary |
| | EN10_WetSeasonShift | Calculates number of weeks delay in the onset of wet season compared to a baseline |

| | | |
|---|---|---|
| Water Quality | EN11_PhytoPlankton | Estimated the phytoplankton growth potential based on empirical relationship with retention time |
| | EN12_AquaticMacrophyte | Estimates aquatic macrophyte growth potential based on empirical relationship with nitrate concentration in irrigation scheme return flow |

***Economic Indicators***

| Category | NB-DSS Script Name | Description |
|---|---|---|
| Navigation | EC1_Navigation | Calculates number of days above baseline flow threshold or change relative to baseline |
| Energy | EC21_AverageEnergy | Calculates average energy generated at specific hydropower node over a specified period |
| | EC22_AverageEnergy_System | Calculates system wide average annual energy |
| Water conservation | EC31_EvapLoss | Calculates average annual evaporation from a dam, a wetland or a lake |
| | EC32_EvapLoss_System | Calculates system wide average annual evaporation |
| Floods | Flood Damage | Calculates flood damage based on damage-depth relationships for different land use types |
| Food production | EC51_FoodProductionSingle | Calculates food production of new irrigation schemes |
| | EC51_FoodProduction | Calculates the potential reduction in crop yield of existing irrigation schemes due to upstream developments |
| | EC51_ProductionIncomeSingle | Calculates actual crop income of new irrigation schemes |
| | EC51_ProductionIncome | Calculates change in crop income of existing irrigation schemes due to upstream developments |

## Expanding the DSS predefined scripts

To expand the DSS predefined set of scripts, you have the following two options:

- Add a new script: In this case you need to do the following:
  - Define what the script will do
  - Identify the data that is needed for the script. This can be external (i.e. data does not exist in the DSS but can be organized in spreadsheets and imported into the DSS) or internal (e.g. generated by DSS Modeling tools).
  - Add the script into a temporary storage within the Script Manager
  - code the script to using Iron Python
  - Test the script to ensure it works properly.
  - If testing is successful add the new script into a storage that is already created into the DSS or if does not fit with any of them create a new storage for it.
  - Ensure the header of the script is updated with a good description.
- Modify an existing script: This option might be needed if you think that the existing script code needs to be improved. In this case you need to do the following:
  - Identify what needs to be changed with the script.
  - Identify if more data that is needed for the script.
  - Modify the existing script[3] code.
  - Test the script to ensure it works properly.
  - If testing is successful add to the modified script into the same storage.
  - Ensure your modifications are added to the header of the script.

## Review Questions

1. What are the main predefined script storages in the DSS?
2. The DSS predefined set of indicators cannot be expanded.
   - True
   - False

[3] It is always advisable to keep a copy of an existing script before modifying.

## Answers

1. The predefined indicators are divided into the following three main categories:
    - BaseUtils.
    - SpreadsheetUtils.
    - IndicatorUtils.
    - NBIScripts.
    - RasterUtils.
    - Environmental Indicators.
    - Social Indicators.
    - Economic Indicators Scripts.
2. False (it can be expanded).

## 2.8. Advanced scripting

### Introduction

This lesson introduces you to two advanced scripting topics, namely, using the DSS Application Programming Interface (API) in scripts (including accessing DSS objects such as time series, GIS layers, scenarios and spreadsheets) and using DSS tools in a script.

Topics covered in this lesson:

- What is the DSS Application Programming Interface (API)
- Using the API to access the DSS objects
- Use the DSS tools in scripts

Lesson objectives:

By the end of this lesson, it is anticipated that you will be familiar with the DSS API and how you can use it to access DSS objects and tools in a script.

### Lesson pre-requisites

You have to be familiar with scripts' basics, complex scripts and Iron Python (See the scripts' basics, complex scripts and IronPython primer sections for details) to take this lesson.

### What is an Application Programming Interface (API)?

An application programming interface (API) is a set of routines, protocols, and tools for building software applications. An API expresses a software component in terms of its operations, inputs, outputs, and underlying types. An API defines functionalities that are independent of their respective implementations, which allows definitions and implementations to vary without compromising each other. A good API makes it easier to develop a program by providing all the building blocks. A programmer then puts the blocks together.

### What is the DSS (API)?

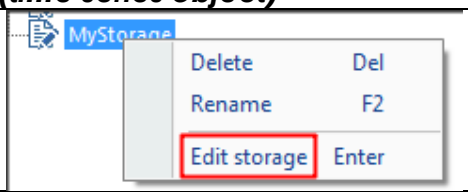
Based on the above definition, The DSS API a set of functions and procedures that allow the creation of applications (e.g. scripts) which access the features or data of the DSS. For example, imagine you need to get time series data from the Timeseries

Manger to calculate its average value. The DSS API should have the functionality that would allow you using Iron Python to do this.

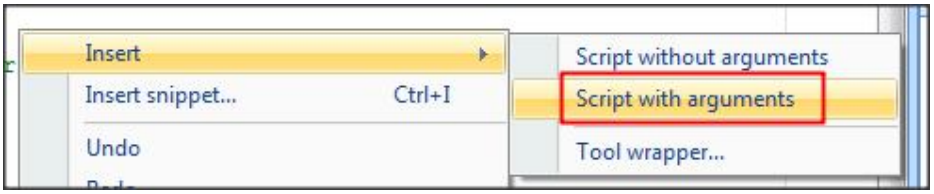
## Exercises

### *Accessing DSS objects using Iron Python(time series object)*

1-Select 'My Storage', right click it and select 'Edit storage'.




2- In the script view, right-click and select 'Insert' then 'Script with arguments' (Note that you can alternatively do this by clicking  the button on the toolbar).




This will insert template code (see below) for a simple script including a header. The script does nothing.

```
def ScriptName(param1, param2):
    """
    <Script>
    <Author>admin</Author>
    <Description>Please enter script description here</Description>
    <Parameters>
    <Parameter name="param1" type="int">Parameter of type int</Parameter>
    <Parameter name="param2" type="IType">Parameter of type IType</Parameter>
    </Parameters>
    <ReturnValue type="IType">Function returns object of type IType</ReturnValue>
    </Script>
    """
    # write your code here
    pass;
```

3- Rename the script and then modify it to calculate the average of a time series. to do this:

- change the function name from 'ScriptName' to 'TimeseriesAverage'
- change the script body to the following:

```
"""
    <Script>
    <Author>admin</Author>
    <Description>Please enter script description here</Description>
    <Parameters>
    <Parameter name="ts" type="IDataSeries">Parameter of
IDataSeries</Parameter>
    </Parameters>
    <ReturnValue type="double">Function returns the average of the
```

```
ts</ReturnValue>
    </Script>
    """
    return ts.Statistics.Average
    pass;
```

Note the changes that was made to the arguments.

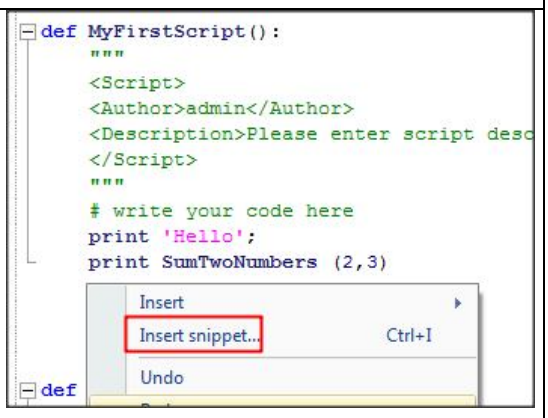
Script should look like the window below

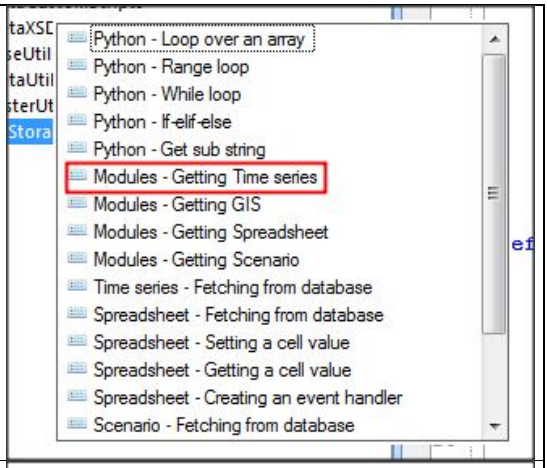
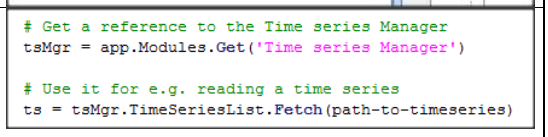
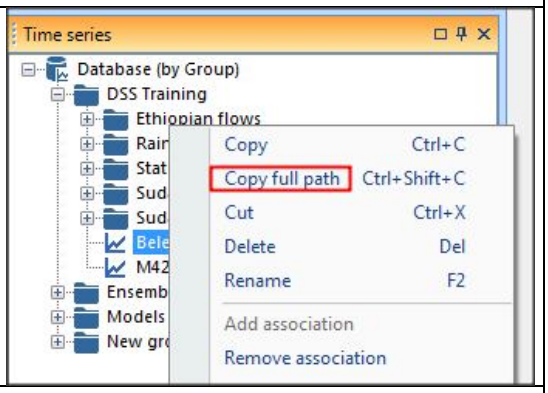
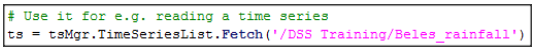
```
def TimeseriesAverage(ts):
    """
    <Script>
    <Author>admin</Author>
    <Description>Please enter script description here</Description>
    <Parameters>
    <Parameter name="ts" type="IDataSeries">Parameter of IDataSeries</Parameter>
    </Parameters>
    <ReturnValue type="double">Function returns the average of the ts</ReturnValue>
    </Script>
    """
    return ts.Statistics.Average
    pass;
```

This script takes one parameter which is a time series. It then uses the Statitics tools of the time series to calculate the average.

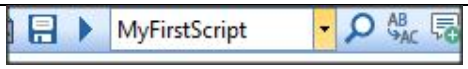
Save the storage by clicking the  on the toolbar and click save to confirm adding the complex script.

4- Now we need to modify the 'MyFirstScript' script to call the 'TimeseriesAverage' script. But to do this we need to have an access to at time series. To do this place the cursor below the last line in the 'MyFirstScript'. right click and select insert snippet.

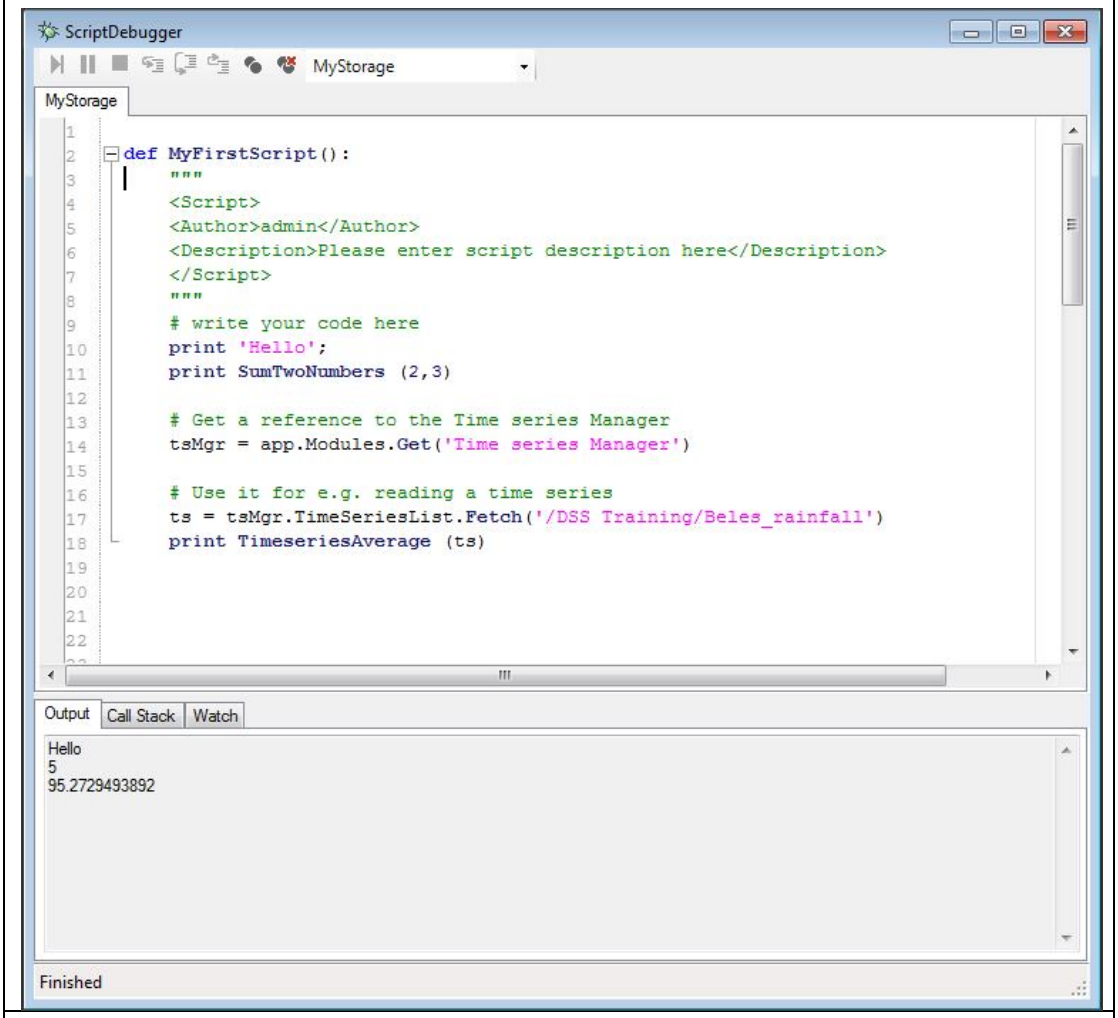


| | |
|---|---|
| 8- The next window appears. It has a slection of code sinppets that can be inserted directely into a script. In this case select the code the gets a time series. |  |
| 9- Examine the lines that were inserted into the script. First an object called 'app' was used to get access to the time series manager (the 'tsMgr' variable). Then, the manager is used to get the Time series data ( the 'ts'). |  |
| 10- Before running the script, you need to define the path to the time series. To do this activate the Timeseries manager and right click a time series and copy its path using the 'Copy full path' option. |  |
| 11- Modify the code as shown next by pasting the full path you copied above and adding a (') before and after the path so it is read as a text. |  |
| 12- Add the following line to call the 'TimeseriesAverage' script:<br>print TimeseriesAverage (ts) |  |

13- Run the 'MyFirstScript' script by selecting it the 'MyFirstScript' in the box and then click the button.




14- The script debugger windows appears and the 'Hello' word appears in the 'Output window' as shown below followed by the sum of 2 and 3 (i.e. 5) and then the average value of the time series.




15- Use the code sinnpets to access other DSS objects such as a GIS layer.

### *Accessing tools using a script*

```
def MyFirstScript():
    """
    <Script>
    <Author>admin</Author>
    <Description>Please enter script description here</Description>
    </Script>
    """
    # write your code here
    print 'Hello';
    print SumTwoNumbers (2,3)

    # Get a reference to the Time series Manager
    tsMgr = app.Modules.Get('Time series Manager')

    # Use it for e.g. reading a time series
    ts = tsMgr.TimeSeriesList.Fetch('/DSS Training/Beles_rainfall')

    print TimeseriesAverage (ts)

    # Get the Resample tool
    tool = app.Tools.CreateNew('Average');
    # Add the time series to the resample tool
    tool.InputItems.Add(ts);

    # Execute the tool
    tool.Execute();

    # Get the output time series
    AvgTS = tool.OutputItems[0];

    print  (AvgTS)
```

## Review Questions

1. What is an API?

## Answers

1. An application programming interface (API) is a set of routines, protocols, and tools for building software applications

# 3. References


- Nile Basin Decision Support System help file (DSS Ver. 2.0)
- Nile Basin Decision Support training material (developed in 2013 and 2014)
- DHI training material for the Nile Basin Decision Support (developed in 2012)
- WP2 Report: NB-DSS WP2 Stage 2 'Data Quality Assurance Guideline: Data Processing, Quality Assurance and Metadata' (2012)
- WP2 Report: NB-DSS WP2 Stage 2 'Guideline for the Evaluation of Water Management Interventions' (2012)