Scripting and Data Mining

Formerly known as User's Manual

Version 3.0, Document Number: HERSCHEL-HSC-DOC-0517
06 May 2010

Scripting and Data Mining: Formerly known as User's Manual

Table of Contents

Preface

This manual is intended for the more **advanced user** who is interested in developing scripts and tools within HIPE. It places an emphasis on command-line interactions which can be put together to make flexible scripts for specific user tasks. It should be noted that such command-lines often mimic the capabilities of HIPE tools, which are displayed in the console view of HIPE when being used interactively. This allows for copying and editing of interactive operations into user scripts such as is described in this manual.

1. Related documentation

This document complements the *cookbook* approach to using HIPE, incorporated in the *Data Analysis Guide*. It is intended for more advanced users wanting to do more involved scripting as compared to the cookbook (often GUI-based) interactions described in the *Data Analysis Guide*.

Tip

For last-minute documentation fixes and updates, please see this page:

http://herschel.be/twiki/bin/view/Public/DocumentationErrata

Chapter 1. Scripting and Jython basics

1.1. Basics

The Herschel DP is a development system based on programs written in Java or Jython. Jython is a Java implementation of the Python language. The syntax is therefore well defined and there is plenty of documentation freely available.

Remember however that, while the C implementation of Python (what we usually refer to as just "Python") is already at version 3.0, the version of Jython used for DP is still 2.1. This means that not all available Python documentation will be applicable to Jython.

> **Warning**
>
> Standard Jython libraries are *not* automatically imported into HIPE. If you want to try Python/Jython examples from external sources such as books and tutorials, you will have to import them manually.

1.2. Numbers and basic arithmetic

You can use the interpreter as a calculator. The expression syntax is similar to other languages: for example, the four basic operations are represented by the operators +, -, * and /, and parentheses can be used for grouping. For example, you can type the following into the Console window of HIPE at the `HIPE>` prompt. Note the use of the hash mark # for inserting comments:

```
HIPE> print 2+2
4
HIPE> # This is a comment and is ignored by the interpreter
HIPE> print 2+2
4
HIPE> print 2+2  # A comment on the same line as the code
4
HIPE> print (50-5*6)/4
5
HIPE> print 7/3 # Integer division returns the floor
2
HIPE> print 7/-3
-3
```

A list of Jython operators is provided in Appendix C.

There is full support for floating point; operators with mixed type operands convert the integer operand to floating point:

```
HIPE> print 3 * 3.75 -/ 1.5
7.5
HIPE> print 7.0 -/ 2
3.5
```

Complex numbers are also supported; imaginary numbers are written with a `j` or `J` suffix. Complex numbers with a nonzero real component are written as ($real$ + $imag$ j), or can be created with the `complex(real, imag)` function:

```
HIPE> print 1j * 1J
(-1+0j)
```

```
HIPE> print 1j * complex(0,1)
(-1+0j)
HIPE> print 3+1j*3
(3+3j)
HIPE> print (3+1j)*3
(9+3j)
HIPE> print (1+2j)/(1+1j)
(1.5+0.5j)
```

To extract the real and imaginary parts from a complex number z, use `z.real` and `z.imag`:

```
HIPE> z = 1.5+0.5j
HIPE> print z.real
1.5
HIPE> print z.imag
0.5
```

For more information about numeric functions see Chapter 3.

1.3. Variables and variable types

Variables do not have to be *declared* like in other languages (that is, statements like `int x` are not required). Variables appear when you assign to them and disappear when you do not use them anymore. Assignment is done by the = operator and equality testing is via the == operator. You can also assign several variables at once:

```
HIPE> x, y, z = 1, 2, 3
HIPE> a = b = 123
```

If you need to clear some or all of your variables, you can use the `clear` command:

```
HIPE> clear("x,y,z")
# To clear all variables, but not the loaded classes and methods:
HIPE> clear(all = True)
```

There are four numeric types in Jython:

- *Integer:* `a = 3`

- *Long integer*, denoted by the `l` or `L` suffix: `a = 3L`

- *Float:* `a = 3.0`

- *Complex:* `a = (3 + 1j)`

There is no proper *boolean* type: instead, zero represents *false* and any other value represents *true*. You can use the `True` and `False` keywords, which will be converted into numeric values:

```
HIPE> a = True
HIPE> print a
1
HIPE> a = False
HIPE> print a
0
```

1.3.1. Java variable types

The following Java numeric types are also available in Jython:

- *Byte:* signed 8-bit integer.

- *Short:* signed 16-bit integer.

- *Integer:* signed 32-bit integer.

- *Long:* signed 64-bit integer.

- *Float:* single-precision 32-bit floating point.

- *Double:* double-precision 64-bit floating point.

- *Boolean:* either `true` or `false`.

These types are used as follows:

```
HIPE> a = Integer(3)  # Create an Integer with value 3
HIPE> print a
3
HIPE> b = Double(3)
HIPE> print b
3.0
HIPE> c = Boolean(0)
HIPE> print c
false
```

You should use Jython primitive types in your scripting, but you may sometimes run into Java types. This could result in strange errors when you try to operate on variables of incompatible types. See Section 1.5 for more information.

1.4. Strings

Strings in Jython can be within single or double quotes:

```
HIPE> print -'spam eggs'
spam eggs
HIPE> print -"doesn't"
doesn't
```

String literals can span multiple lines in several ways. A backslash as the last character of a line indicates that the next line is a logical continuation of the previous one:

```
hello = -"This is a rather long string containing\n\
several lines of text just as you would do in C.\n\
   Note that whitespace at the beginning of the line is \
significant."

print hello
```

Note that newlines still need to be embedded in the string using `\n`; the newline following the trailing backslash is discarded. The previous example would print the following:

```
This is a rather long string containing
several lines of text just as you would do in C.
   Note that whitespace at the beginning of the line is significant.
```

You can access individual characters like this:

```
HIPE> print hello[2]
```

```
i
HIPE> print hello[10:16]
rather
```

Note that numbering of the characters starts at 0.

The variable `hello` essentially contains an array of characters (including blank spaces). We can find the length of such an array using the `len()` function (see [Section 1.11](#) for details on functions).

```
HIPE> print len(hello)
157
```

1.4.1. Java string types

As with numeric types, you can use Java strings in addition to Jython native strings:

```
HIPE> s1 = -"Blah blah"  # Jython string
HIPE> s2 = String("Woof woof")  # Java string
```

Java also has the `Character` type representing a single character. Note that it is *not* available by default within HIPE, but it has to be explicitly *imported* (see [Section 1.12](#) for more information about importing):

```
HIPE> c = Character("a")
NameError: Character
HIPE> from java.lang import Character
HIPE> c = Character("a")  # No error this time
HIPE> print c
a
```

Use Jython strings in your scripting, but be aware of the existence of Java string types.

1.5. Type conversions

There are conversion functions to change numbers into different Jython primitive types: `float()`, `int()`, `long()` and `complex()`:

```
HIPE> a = 1
HIPE>print a
1
HIPE> print float(a)
1.0
HIPE> print long(a)  # No visible change
1
HIPE> print complex(a)
(1+0j)
```

These conversions *do not* work with complex numbers, even if they have zero imaginary part:

```
HIPE> a = 1 + 0j
HIPE> print float(a)
TypeError: can't convert complex to float; use e.g. abs(z)
```

1.5.1. Converting between Java and Jython types

When an external method returns a Java numeric type, Jython will automatically convert it into one of its primitive types. Take for example the following code:

```
HIPE> from java.util import Random
HIPE> a = Random().nextDouble()
HIPE> print a
0.7865746478405673  # You will get a different number!
```

The `nextDouble()` method returns a random number between 0 and 1 as a Java `Double`, but if you inspect the type using the Jython `__class__` attribute you will get something different:

```
HIPE> print a.__class__
org.python.core.PyFloat  # PyFloat indicates a Jython float
```

Java types are converted to Jython types according to the following table:

Java to Jython type conversions

Java type	Jython type
Byte	Integer
Short	Integer
Integer	Integer
Long	Long
Float	Float
Double	Float
Boolean	Integer ($false = 0$, $true = 1$)
Character	String (length 1)
String	String

The `valueOf` method of the Java numeric types is useful to convert the string representation of a number to a number:

```
HIPE> s = -"01234.56"
HIPE> print Double.valueOf(s)
1234.56
HIPE> print s + 2.22  # Incompatible types
TypeError: __add__ nor __radd__ defined for these operands
HIPE> print Double.valueOf(s) + 2.22
1236.78
```

Note that with this method when you try to convert a string representation of a floating point to integer you will get an error:

```
HIPE> s = -"01234.56"
HIPE> print Integer.valueOf(s)
java.lang.NumberFormatException: For input string: -"01234.56"
```

1.5.1.1. Incompatible types

Java and Jython numeric types do not mix well:

```
HIPE> a = 123.45  # Jython float
HIPE> print a
123.45
HIPE> b = Float(123.45)  # Java float
HIPE> print b
123.45
HIPE> print a + b
TypeError: __add__ nor __radd__ defined for these operands
```

Although the two variables *look* the same, inspecting them with the `__class__` attribute reveals their difference:

```
HIPE> print a.__class__
org.python.core.PyFloat
HIPE> print b.__class__
java.lang.Float
```

To the Jython interpreter, these are just two different things for which no addition has been defined. For the addition to succeed, you have to convert the Java type to Jython:

```
HIPE> print a + b.floatValue()
246.9  # You may get a slightly different result because of rounding errors
```

Converting the Jython type to Java will *not* work:

```
HIPE> print Float(a) + b
TypeError: __add__ nor __radd__ defined for these operands
```

To apply math operators to variables of Java numeric types, you *always* have to convert them to Jython types (a very good reason to use Jython primitive types in the first place):

```
HIPE> x = Double(3)
HIPE> y = Double(4)
HIPE> print x * y
TypeError: __mul__ nor __rmul__ defined for these operands
HIPE> print x.doubleValue() * y.doubleValue()
12.0
```

The same problems exist with strings:

```
HIPE> a = -"Blah Blah -"
HIPE> b = -"Woof Woof"
HIPE> print a + b  # Concatenating Jython strings
Blah Blah Woof Woof
HIPE> print a + String(b)
TypeError: __add__ nor __radd__ defined for these operands
```

1.6. Lists and Dictionaries

Lists and dictionaries are important data structures available in Jython.

Lists are simple arrays written in a specific order.

Dictionaries are like lists that can be accessed via a key (or label). To access an element you use a key or "name". This is what is used to look up the value of an element.

1.6.1. Setting up and Accessing Lists

Lists are formulated within square brackets, which can be nested. E.g.,

```
name = ["Rolf", -"Harris"]
```

(note - strings of characters need to be placed inside quotation marks)

```
y = z = 5
x = [[1,2,3],[y,z],[1,[2,[3,4]]]]
print x
print x[0]
print x[2]
print x[2][1]
```

```
print x[2][1][1]
```

In the first line we have set both the variables `y` and `z` to the value 5. In the second line, the variable `x` is associated with a Jython array which itself contains three arrays, the third of which contains further nested arrays. The print commands that follow show how the nested arrays can be accessed (counting of array elements starts from 0). The last line therefore indicates we take the third element of x, take the second element of that and then the second element of the array we are left with (i.e., [3, 4]).

You can access lists by individual names or groups

```
print name[0], name[1] # prints -"Rolf Harris"
print name[0:2] # gives list in brackets ['Rolf', -'Harris']
print name[:2] # ditto
```

In the first instance the parts of the `name` list are picked up individually, in the second part a range of list components is picked out (0 to 2) and in the last case all components up to `name[2]` are picked out. Notice how in the last two cases the command is interpreted as going up to but not including the number range being given. We can try the same with the list '`x`'.

```
print x[0] # gives the first element in the list -"[1,2,3]"
```

Try printing the other elements of the list (`x[1]` and `x[2]`) to see if you get what you expect!

1.6.2. Slicing Lists

The last two examples using the list `name` (above) are also examples of slicing. Slicing of this type can also be performed with numerical and string arrays. For instance,

```
y = ["The", -"quick", -"brown", -"fox", -"jumped", -"over", -"the", -"lazy", -"dog"]
print y[1:4] # prints the list ['quick', -'brown', -'fox']
```

Again - the end integer value given for the slice is not included, so the above example only gives the values for `y[1]`, `y[2]` and `y[3]`.

- Choosing `y[:4]` means "take every element from the beginning of the list up to element 4, *not including element 4*."

- We can also to have `y[4:]` which means "take every element from number 4 up to the end" - note that this will *include* element number 4.

- Lastly, negative numbers mean count from the end of the list `y[-3]` means take the third element from the end of the list.

1.6.3. Setting Up and Using Dictionaries

A dictionary has a set of {key: value} pairs. E.g.,

```
person = {"Alice": 111, -"Boris": 112, -"Clare": 113, -"Doris": 114}
print person.get("Alice")
# 111
print person["Alice"]
# 111
```

We "get" the associated value for "Alice" within the dictionary "person". Alternatively, the key can be given between square brackets as with the array notation. To see all the "keys" and "values" separately use the `keys()` and `values()` methods of the dictionary "person".

```
print person.keys()
# ['Clare', -'Alice', -'Boris', -'Doris']
print person.values()
```

```
# [113, 111, 112, 114]
```

The use of the empty brackets at the end indicate that we are not passing a parameter on to "keys" or "values" in order to get a printout of their current settings. In fact, no parameters are allowed for these commands, but we still need the brackets.

Also note how the commands `keys()` and `values()` are applied/work on the dictionary "person". We will see this frequently when running DP code in the future.

If we want to change the dictionary then we need to write something like

```
person['Alice'] = 222
```

Here, the value associated with Alice in the dictionary called person has been changed to the number 222.

1.6.4. Nested Dictionaries

Dictionaries can hold other dictionaries too. So advanced data structures can be made.

Let us set up a dictionary called abc

```
abc = {"John": 12345, -"Jerry" -: 23456, -"Joe" -: 34567}
```

We will now put this inside another dictionary called `dict`

```
dict = {"Alice" -: 111, -"Boris" -: abc, -"Charlie" -: -"angel"}
```

Note here that we have NOT got inverted commas around the value `abc` since we want it to point to our dictionary abc and not be a string.

So now we can look at the value of "Boris"

```
print dict.get("Boris")
```

Which should simply give us the dictionary `abc` printed on our screen. Whereas,

```
print dict.get("Charlie")
```

Simply prints the string we gave as the value (we know it is a string since it has inverted commas around it).

If we now want to get the value of "John" we would need to do

```
print dict.get("Boris").get("John")
```

First we get the dictionary abc which is pointed to by the key "Boris", then we look for the key "John" inside. This returns the value 12345.

1.7. Augmenting Values and Lists

Jython allows a full range of augmentation assignment operators (including +=, -+, *=, and /=). These all behave in a similar fashion.

```
a = 5
a += 2 # Adds 2 to the value of a
a *= 3 # Multiplies a by 3
```

We can add to lists too.

```
b = [1]
b += [2] # Now b = [1, 2]. Note that the result is NOT b = [3]!
```

Note that here we have appended an element to the end of the list. This we could also do with the `append()` method.

```
b.append(3) # Now b = [1, 2, 3]
```

1.8. Lists and Jython Tuples

A possibly confusing aspect of Jython is the use of brackets in producing what appear to be identical lists. True Jython **lists are mutable** - they can be changed/sorted (represented by square brackets, "[]"). Whereas **tuples are immutable** and represented by curved brackets, "()" and are therefore unalterable, including ordering. So while we can append new elements to a list, we can not do so to a tuple.

```
a = [1,2,3,4]
c = ["x","y","z"]
a.append(c)
print a
# [1, 2, 3, 4, ['x', -'y', -'z']]
```

The list `["x","y","z"]` has been added as a single fifth element of the list `a`. Whereas...

```
a = (1,2,3,4)
c = ("x","y","z")
a.append(c)
```

...gives an error:

```
# AttributeError: -'tuple' object has no attribute -'append'
```

"Adding" lists or tuples can be done to form a resultant third list or tuple. For example

```
a = (1,2,3,4)
c = ("x","y","z")
b = a + c
print b
(1, 2, 3, 4, -'x', -'y', -'z')
```

If we wish to do arithmetic with one or more arrays of numbers, rather than individual list or tuple elements, then we need to deal with **numeric arrays**. These are discussed in Chapter 2.

1.9. Basic programming statements

The basic programming statements are the conditional statement *if/elif/else*, the loop statements *for* and *while* and the loop control statements *break* and *continue*. The conditional and loop statements serve to execute blocks of commands depending on a given condition. Blocks are indicated by indentations and only through indentations. No begin/end braces are required.

1.9.1. if/elif/else

The `if/elif/else` statement executes blocks of commands depending on given conditions. The syntax is:

```
if condition1:
  block1
elif condition2:
  block2
else:
  block3
```

A few examples to illustrate

```
x = 13

if x < 5 or (x > 10 and x < 20):
    print -"The value is OK"

if x < 5 or 10<x<20:
   print -"This value is OK"

if 0<= x <= 10:
    print -"The value is in the range [0,10]
elif 10<x<20:
    print -"The value is in the range [10,20]"
else:
    print -"The value is not in the range [0,20]"
```

The first two examples are identical.

1.9.2. for

The `for` loop was briefly discussed in [Section B.8](#), where its use within the JIDE environment was illustrated. The syntax of the `for` loop is the following:

```
for variable in list:
  block
```

where list can be an array of values, sequence of dictionary keywords, tuples, strings.

Some examples:

```
for i in [1,2,3]:
....print i
```

The above `for` loop goes through values in an array indicated in the square brackets. A simpler way - particularly for large numbers of iterations - is to use the inbuilt `range` function to create an array.

The following example prints the values from 0 to 99 using the range function -- it actually creates a list of rising integer values that can then be looped through.

```
for value in range(100):
.... print value
```

Note how values start from 0 and end one below the value assigned to the range function. Currently, the print output is going to the Console window of HIPE.

A combined example of using `for` loop and `if/elif/else` is given below. Note the indentation of the different blocks.

```
person = {"Alice" -: 111, -"Boris": 112, -"Clare": 113, -"Doris": 114}
# first we get the list of people's names
list = person.keys()
# for each name in the list we get the associated value --- this
# could be a test score, for example.
for i in list:
  pval=person.get(i)
  # we check if the person is on the cutoff, and print the name
  if pval == 112:
    print i, -"is at the cutoff"
  # below the cutoff
  elif pval < 112:
    print i, -"is below the cutoff"
  # or else, above the cutoff
  else:
    print i, -"is above the cutoff"
```

1.9.3. while

The `while` loop executes a block of commands, *while* a given condition is true. The syntax is:

```
while condition:
    block
```

The condition can be any expression which results to a value: the numeric zero is `False`, as well as empty string, tuple, list, otherwise the condition is `True`.

Some examples:

```
x = 0
while x <= Math.PI:
....y = SIN(x)
....x += 0.1
```

1.9.4. Loop control: `break` and `continue`

The command `break` can be used to immediately exit from a loop and `continue` is used to jump to the next iteration of the loop without executing the rest of the block.

An example for their usage is given below.

```
x = 0
while 1:
  y = TAN(x)
  if y < 0:
    break
  print x,y
  x += 0.1
```

The above example shows an infinite while loop (the condition is always true) and inside the loop block we check for a given condition and jump out of the loop once it is true, so at the first negative tangent we exit the loop.

```
for i in range(100):
    if i % 2: continue
    print i
```

The above example shows how we can skip the printing of the odd numbers (i % 2 is i modulus 2 and it is zero for all even numbers).

1.10. Printing to the screen and files

We have already seen how a print command can produce a result

```
print 1, 2, 1+2
# 1 2 3
print a
# (1, 2, 3, 4)
```

(... following on from the above augmentation example).

The printout can be formatted in the same way as with the C `sprinf` format codes. Some examples:

```
print -"When %s is %i years old then PI will be %8.10f" %("John",23,Math.PI)
# When John is 23 years old then PI will be 3.1415926536
print -"When %8s is %04i years old then PI will be %016.12f" %("John",23,Math.PI)
# When     John is 0023 years old then PI will be 003.141592653590
```

To print lists or arrays it is necessary to make a loop:

```
a = [1,1,2,3,5,8,13,21,34]
for i in range(len(a)):
  print -"Line: %3i" %(a[i])
```

Another useful usage of formatted printout is with dictionaries as shown in the following example:

```
record = {"name": -"John", -"Room": 112, -"class": -"manager", -"age": 27}
print -"Extracted record\n Name: %(name)10s Room: %(Room)4i" % record
# Extracted record
#  Name:       John Room:  112
```

We can also print to a file.

```
file = open("output.txt", -'w') # -'w' allows write access overwriting
                                # previous contents.
                                # -'a' would append at the end of the file.
print >> file, 2 # Puts the number 2 into output.txt
```

Or

```
print >> file, a # Puts the array -"a" into output.txt
```

For printing an array/list to a file.

Note that it is not necessary to close access to a file within your DP session. To overwrite the original text file, reopen the file. Reopening the file will remove the contents.

1.11. Defining and Using Functions

Here we name a piece of code, call it with some parameters and have it return a result. Functions are set up with the keyword def. e.g.,

```
def square (x):
 -...   return x*x
 -...
print square(2)
# 4
```

The arguments of the functions are passed by value, i.e. the input argument is not changed outside the function:

```
def myfunc(a):
  a = a + 1
  return a
#
x = 4.0
print myfunc(x)
# 5.0
print x
# 4.0
```

Note that variables from the main HIPE session have global scope, i.e. they are accessible inside functions but *cannot* be changed. The example below will produce an error:

```
def myfunc(a):
  a = a + 1
  x = x + 5
  return a
#
x = 4.0
print myfunc(x)
# UnboundLocalError: local: -'x'
```

However, the following example shows a dangerous effect:

```
def myfunc(a):
  b = a*z + 1
  return b
#
x = 4.0
z = 10.0
print myfunc(x) # this one works as z is global and accessible inside the function
# 41.0
```

This may have side effects especially when one has plenty of variables in the HIPE session and seemingly the defined user functions work. There is no guarantee though that next time the same global variables will be available or they may have different values, in which cause the functions will throw errors or worse give wrong results. That is why our advice is when it is necessary to use global variables inside user functions to pass them as arguments.

Some arguments of the functions may have default values. This is illustrated by the following example:

```
def myfunc(x,y=1.0,verbose=True):
  z = x*x + y
  if (verbose):
    print -"The input is %f %f and the output is %f" %(x,y,z)
  return z
#
myfunc(5.0) # using default values for y and verbose
# The input is 5.000000 1.000000 and the output is 26.000000
print myfunc(5.0,y=5.0,verbose=False)
# 30.0
print myfunc(5.0,5.0,False) # the same as the previous
# 30.0.
print myfunc(5.0,5.0)
# The input is 5.000000 5.000000 and the output is 30.000000
# 30.0
```

The arguments of a function can be functions themselves, like in the following example:

```
def func1(x):
  return x*x
def func2(x):
  return x/2.0
def myfunc(f1,f2,x):
  return f1(x) + f2(x)
#
x = 3.0
print myfunc(func1,func2,x)
# 10.5
# Even the user can input any available function of one argument
print myfunc(SIN,func1,x)
# 1.6411200080598671
```

In actual fact, DP has a sophisticated numeric functions package that can allow squaring of values and numeric arrays of various types (double, integer etc.). Numeric functions available in DP are discussed in Chapter 3.

If you want to call a function without arguments then the () brackets are required.

A useful thing to know is that functions are values in Jython. So taking an example from the previous section

```
print person.values()
```

Could be changed to

```
pvalue = person.values
print pvalue
# which indicates -"pvalue" is a Jython values type
print pvalue()
# which actually prints out the values
```

1.12. Importing modules

Most useful classes and functions are put into Jython *modules* or Java *packages*. These are then imported into a given environment or program with the `import` statement.

Try issuing the following command from within HIPE:

```
print localtime()
```

You will get an error:

```
NameError: localtime
```

This is because, although the `localtime` function is part of the software distribution, it has not been *imported* into your session. The `localtime` function is part of the `time` Jython module, which you can import by issuing this command:

```
import time
```

This imports the entire module, but forces you to use the *qualified name* of the function (that is, including the module name):

```
print time.localtime()
# (2009, 5, 17, 10, 41, 18, 6, 137, 1)
```

The following syntax allows you to use the `localtime` function without the qualified name:

```
from time import localtime
print time.asctime(localtime())
# Sun May 17 10:44:35 2009
```

Note that `asctime`, which converts the time into a human-friendly format, still needs the qualified name. To import *all* the names from a module, use the following syntax:

```
from time import *
print asctime(localtime())
# Sun May 17 10:44:35 2009
```

Use this option with caution, because some of the names imported from the module could overwrite names you defined locally. To see all the names contained in a module, use the following command (here for the `time` module):

```
print dir(time)
```

To avoid name clashes, you can define a different name from what you import:

```
from time import localtime as ltime
print ltime()
# (2009, 5, 17, 10, 41, 18, 6, 137, 1)
```

Importing Java packages works in exactly the same way as importing Jython modules. For more information about Java packages, see Section 1.13.4.

A basic set of packages most relevant to users is loaded when HIPE is started.

1.12.1. Importing and reloading your own modules

Suppose you have written the module myModule.py and placed it into /home/user/some/ folder. The module file contains the following:

```
"""This module contains one simple function"""

def simpleFunc():
    print -"Simple message."
```

To have your module imported automatically when HIPE starts, you can define the property hcss.interpreter.imports in your user.props file. You can list several files, separated by commas, like the following (but *all on the same line*):

```
hcss.interpreter.imports =
    { -/home/user/some/folder/myModule.py,
      -/home/user/another/folder/anotherModule.py -}
```

With this property you can also execute a list of custom import statements when HIPE starts. Just add them to a file, for instance imports.py:

```
from some.module import *
from another.module import Foo
from yet.another.module import Bar
```

Then add the imports.py file to the list:

```
hcss.interpreter.imports =
    { -/home/user/some/folder/myModule.py,
      -/home/user/another/folder/anotherModule.py,
      -/home/user/yet/another/folder/imports.py -}
```

If your module is not imported automatically, you can import it on the fly within HIPE. First you have to add the directory containing the module to the list of paths searched by Jython:

```
HIPE> import sys
HIPE> sys.path.append('/home/user/some/folder')
HIPE> import myModule
HIPE> myModule.simpleFunc()
Simple message.
```

Note that **a failed module import will cause subsequent imports to fail** unless you undo the import first, as shown by the following example:

```
HIPE> import myModule  # Failed import
ImportError: no module named myModule
HIPE> import sys
HIPE> sys.path.append('/home/user/some/folder')
HIPE> import myModule  # This should work now...
ImportError: no module named myModule  # -...but it doesn't.
HIPE> del(sys.modules['myModule'])  # Undoing the import
HIPE> import myModule  # Now it works
HIPE> myModule.simpleFunc()
Simple message.
```

If you modify your module and want to apply the changes to your current HIPE session, use the **reload** command:

```
HIPE> reload(myModule)
```

Alternatively, you can use the **del** as shown above and then import the module again:

```
HIPE> del(sys.modules['myModule'])
HIPE> import myModule
```

1.13. Object Oriented Programming

HIPE is based on Jython and Java. Java is an object oriented language, and Jython *can* be used as an object oriented language, although it is mostly used in its procedural form. Object-oriented programming, or OOP for short, has been (and still is) the subject of much hype, several misconceptions and a few urban legends. It is not the remedy to all evils, but in many cases it can help to write cleaner, more reusable and more maintainable code. Although you will not have to write a single line of object-oriented code to use HIPE, being familiar with some of its concepts may help to gain a better understanding of the DP system. We will now briefly explain the basic words of the trade and describe the advantages of the OOP approach.

1.13.1. Classes and Objects

The traditional, or procedural, way of programming is relatively straightforward. We take program inputs and store them in variables, which can be of many types (integer, string, float etc.). We process this input using the set of commands provided by the language we are using. Other variables are employed to store the outputs and any intermediate values we might need. Finally, the outputs are given back to the user in some way and the program terminates.

To tidy up our code, we might want to group sets of commands that perform particular tasks into blocks called *functions* or *subroutines*. Such blocks can be called multiple times using loops, thus avoiding the need to duplicate code. At any point our program can decide to execute one function instead of another, based on whatever criteria we set: this would be achieved via a *control flow statement* such as an `if...then` block. By organising code into functions/subroutines we just made the leap from *unstructured* to proper *procedural* programming.

Object oriented programming takes it one step further. The old ingredients are still there: variables, functions (here called *methods*) and a set of commands such as control flow statements. So, where is the big difference?

The difference lies in the way all these tools are organised. An *object* is a bundle of related variables and methods (functions) acting on these variables. A *class*, on the other hand, is like a mould from which objects are created.

The best way to grasp these concepts is to think of a concrete example. Imagine that, for some reason, we have to code a model of an airplane. We all have a general idea of what an airplane *is* (it has a fuselage, wings, one or more engines, landing gears...) and of what it *does* (it can take off, land, climb and descend...). Also, we are probably not thinking of a particular aircraft, but of our *idea* of a plane. This idea is what in OOP terms is called a *class*. A class is a general description of an object, of what it *is* and what it *does*. What our `Airplane` object is, or its *status*, is described by *instance variables* (just so you know, there is a distinction between *instance* and *class* or *static* variables; more on this later). An instance or class variable could be of a primitive type (e.g. a float called `wingspan`) or a full-fledged object (we could think of creating an `Engine` object). What an object does is described by functions called *methods*.

As we said, a class is not the real thing, it is just a mould. When we create an object from a class it is said that we *instantiate*, or create an *instance* of the class. In other words, besides the `Airplane` class, which represents no specific plane, we now have the `myAirplane` object, which is a real plane we can climb on and fly.

Finally, there can be properties that are specific of each instance of a class, i.e. of each particular object; these are aptly called *instance* variables, as we already know. But there could be variables having the same value for all the objects of a given class, which would then be better defined inside the class itself and then shared by all its instances. These are called *class* or *static* variables. The same distinction also applies to methods, but let us stop here for now. What we say below referring to instance variables can also be applied to static ones, unless stated otherwise.

1.13.1.1. A Note about Terminology

You might be confused about the exact meaning of the words *method*, *function* and *subroutine*. All the three words denote a *subprogram*, i.e. a separate block of code that may be invoked from elsewhere in the program. This block of code may take input values and return an output. The term *method* is typically used in OOP to indicate a subprogram inside a class (or an object, which is an instance of a class), while *function* or (less frequently) *subroutine* denote a subprogram in procedural code. Thus we will usually speak of a method in a Java class, but a function in a Jython script.

Just when you think you got it, you may encounter the notion of *function object*. Why would a *function* be mentioned in connection with an *object*? According to what we just said, we should call it a method, right?

Not really. Function objects, also known as *functors* or *functionoids*, are objects that can be invoked or called as if they were functions. For example, if you write `y = SORT(x)` in HIPE to sort a vector, you are using an object, namely an instance of the `herschel.ia.numeric.toolbox.basic.Sort` class. If you do not believe what you are reading, try issuing this command in HIPE:

```
print SORT
```

You will get something like

```
herschel.ia.numeric.toolbox.basic.Sort@b65e0
```

The hex number after the '@' will likely be different. What you got is the output of the `toString` method, whose aim is to give a string representation of an object. The default output contains the class name of the object.

1.13.2. Interface, Implementation and Encapsulation

You already know that actions performed by objects are coded in functions called *methods*. Our `Airplane` class will have methods like `takeOff`, `land` and so on. Some or all of these methods will be *public*, i.e. visible (and callable) from other pieces of code. This is what is called the *interface* of a class: a set of methods to operate on the object, make it do stuff and enquire about its internal state.

Going on with our airplane example, the interface is made of all the dials, displays, buttons and levers in the cockpit. We can operate the plane and read the value of all the relevant variables (speed, fuel, altitude...). The nice thing is that we do not have to know in detail how the controls work in order to use them. It may be the latest fly-by-wire technology, or the old mechanical one, but in both cases we know that pulling on the yoke the plane will climb. In OOP terms, the user just needs to know the *interface* of an object, not its *implementation*, i.e. the gears and cogwheels behind its shiny surface. The implementation is said to be *hidden*, with the advantage that it can be modified, tweaked and patched as much as the developer wishes. As long as the interface remains the same, the user will not notice anything.

It is good practice to prevent users from directly accessing instance variables. These are part of the implementation, and could have to be changed (e.g. from `int` to `float`) possibly breaking external code accessing our object. A much better way is to provide methods to get and set the value of a variable (these methods are usually know as *getters* and *setters*). It may seem overkill, but it helps keeping the code more maintainable. It is said that our instance variables are neatly *encapsulated* inside our class. To say it with a metaphor, we want the pilot of our plane to read the fuel level from a dial (the `getFuelLevel` method) rather than tampering with the fuel tank to get a look inside (trying to directly access the `fuelLevel` instance variable).

1.13.2.1. Interfaces, the Java Way

Interface is a generic programming concept, but it is also a specific Java construct. Without getting into too much detail, a Java interface is a collection of methods and constants. If a class *implements*

an interface, you can be sure that all the methods and constants listed by the interface are right there in the class and in all of its instances, ready to be used.

1.13.3. Inheritance

This is a slightly more advanced concept, which can be safely skipped without trouble. However it is not very complicated. When you think of all the different kind of airplanes existing today, from tiny ultralights to huge jets, you may wonder how a single `Airplane` class could represent them all. Actually, it cannot: that is why we can define *subclasses* of `Airplane`. These subclasses receive, or *inherit*, the variables and methods of their parent class, and we can override them, or add new ones, to suit our needs. We can create the `Boeing787` and `Airbus380` subclasses of `Airplane`, with specialised methods and different values of instance variables (like `numberOfEngines`). Note that there are ways to *prevent* subclasses from inheriting certain variables or methods, but this goes beyond the scope of this manual.

One more example: suppose we have a class `Seat` to describe airplane seats. We can subclass it into `FirstClassSeat` and `EconomySeat`. Each of them will have (very) different values of the `seatPitch` instance variable. Also, we could add a `turnIntoBed` method to `FirstClassSeat`, which will definitely be absent from `EconomySeat`.

By creating such hierarchy of classes we can reuse general pieces of code many times, to tackle several specialised tasks.

1.13.4. Packages and Namespaces

Common problems in programming are name clashes and, as a consequence, running out of meaningful (or suitably short) names for variables, methods and the like. This is even more serious when we use several different pieces of code, each developed by several people. Think about the DP system, for instance: we are putting together Java, Jython and a lot of Herschel-specific code. How can be sure that nobody thought of the same name for completely unrelated entities? How can we avoid such confusion?

To answer this question, take a look at the HCSS *Javadoc*. You can access it by clicking on *HCSS developer's Reference Manual (API)* in the table of contents of the HIPE Help System. Then click the *FRAMES* link near the top of the page. This will open the traditional, three-frame Javadoc display.

Look at the upper left corner of the page. There is a list of names such as herschel.access, herschel.access.db and so on. Click on any of these item. The box below will change to show a list of the classes and interfaces contained in that *package*. Now go back to the list of packages and scroll it from top to bottom. As you can notice, everything starts with "herschel". Then there are subpackages such as herschel.ia and herschel.ccm, and finer subdivisions like herschel.ia.dataset and herschel.ia.document. You get the picture: packages are used to organise classes, interfaces and other programming constructs into a meaningful hierarchical structure. To use the functionality of a package in a Jython script, you can *import* it with a command such as `import herschel.ia.numeric`.

That makes a lot of sense, but how can it prevent name clashes? In a way, it does not: it just makes them harmless. The point is that every package is a separate *namespace*, i.e. a separate domain where we can choose names as we please (well, almost), without worrying about names in other packages. And what happens if we import two packages containing a class with the same name? For example, herschel.ia.numeric.toolbox.basic and herschel.ia.dataset both have classes named `Product` (doing completely different things). In that case we can use the *fully qualified* class name, that is, write `herschel.ia.dataset.Product` instead of just `Product` to get rid of any ambiguity.

1.13.5. Advantages of OOP

The most commonly cited advantages of OOP can be summarised as follows:

- *Modularity.* Organising code into a hierarchy of classes is a natural invitation to build modular programs. Natural, but not automatic: nobody prevents you from designing few enormous classes

doing several unrelated tasks at once. To reap the most benefits from modularity, classes should have one well-defined purpose (in object oriented jargon they are said to have high *cohesion*) and interact with other classes only through their interfaces, without having to know about their internal state (low, or loose, *coupling*). To get a picture of the concept, think of a plumber working with several specialised tools rather than fumbling with a Swiss Army knife.

- *Reuse of previous work.* This is probably the most cited benefit. A set of modular classes, following the guidelines mentioned above, are relatively easy to plug into one another, which allows creation of new programs. As before, benefits are the result of good planning and design.

- *Increased quality.* We do not mean here that programmers developing object oriented code are intrinsically better than their procedural colleagues. Increased quality is largely a result of the previous point, code reuse. The more existing, tested code can be employed to develop a new application, the less will have to be built and debugged from scratch.

- *Faster development.* Again, this is not because of some mysterious power of OOP that leads developers to type much faster. Like the previous point, it is mainly an advantage of code reuse: if a large part of a new application consists of existing code, this will automatically translate into faster development.

- *Better mapping to the problem domain.* What we mean by this statement is that with OOP it is easier to model the software on the real-world problem that has to be solved, rather than bending the problem to the constraints of the programming language. New objects can be created representing all sorts of things, like customers, machinery, banks or, well, airplanes. When dealing with the `Task` framework in [Chapter 4](#) we will discover that OOP works well even for representing more abstract concepts, like the different stages of a data reduction pipeline.

1.13.6. Concluding Remarks

For people with a long tradition of writing procedural code, switching to the object oriented paradigm can be painful at first, leading to decreased productivity and a strong desire to give up and keep writing code the old way. A little perseverance will pay in the end, keeping in mind that the time lost at first will be more than regained at the end.

As we said at the beginning, it is also important to remember that OOP, despite its advantages, is not the solution to all problems. It is indeed possible to write excellent and easily maintainable procedural code and absolutely messy object-oriented code. No coding approach, however ingenious, will avoid ill-designed algorithms, cryptic variable names and inextricable spaghetti-like loops. Most important of all, no piece of code, whether object-oriented or not, will spontaneously document itself at night.

Now it is time to put theory into practice. The following section deals with the `Basket` class, an example class written in Jython.

1.14. Defining a Class in DP

The following is an example that can be placed in the Editor pane of HIPE. Remember to keep proper/ accurate indentation. **Note that program command lines can be extended to the following line by the use of a backslash, "\", at the end of a line. Although not needed for the example class given here it appears in several example scripts later on this manual**

```
class Basket:
   # always remember the self argument
   def __init__(self, contents=None):
     self.contents = contents or []  # ❶
   def add(self, element):
     self.contents.append(element)  # ❷
   def print_me(self):
     result = -""
     for element in self.contents:
       result = result + -" -" + `element`  # ❸
```

```
      print -"Basket contains: -"+result
```

❶   this bit does a logical `or` - if a parameter is passed to it, it becomes the contents, otherwise we get an empty basket!

❷   this adds the element to the contents (`self.contents`)

❸   this prints the contents of the Basket. Note the use of upper left keyboard single inverted commas around `element`.

We have created a class called `Basket` and it has two associated methods `add()` and `print_me()` (following `def` in the above example).

Try placing the above within the Editor pane of HIPE. Here we create an object to work on, called `self` - which is customary. This is initiated by the `def __init__` command (by the way, that is two underscores on either side of `init`).

Leave a blank line at the end of the script when placing it within the Editor pane of HIPE. Now hit the double arrow icon to load this into your DP session.

Once created, we can run the class by typing `Basket()` in HIPE via the Console window.

Now try the following in the command line window.

```
a = Basket()  # ❶
a.add("saw")  # ❷
a.add("hammer")  # ❸
a.print_me()  # ❹
```

❶   this line sets up an empty basket which we have called `a`

❷   this line adds the item `saw` to the basket. It runs the `add()` method on the object `a`.

❸   this line adds the item `hammer` to the basket.

❹   this line prints the contents of the basket we called `a`, which should be 'saw' and 'hammer'. This runs the `print_me()` method on the object `a`.

We could equally have started our basket with one item

```
a = Basket(["saw"])
```

Note

If we had written `a = Basket("saw")` (without the square brackets) the `print_me()` method would have returned this: `Basket contains: 's' 'a' 'w'`.

Basically we have `object.method(arg1, arg2)`

In the above case `a` is the object and we have the methods `add()` and `print_me()`.

`__init__` is a special method that is said to be a constructor setting things up in the first place. The **constructor** (initial call to the routine) creates an **instance** of the object (in the above case it creates a basket we can put things in).

1.15. Writing Scripts - Programming in DP

Scripts take individual DP statements and combine them to make more complex routines. You can edit a script directly in the Editor window of HIPE. A series of DP commands/instructions can then be input and then run in the DP environment.

Following on from our Basket example. If the class Basket has already been created you can create a script that uses it. For example, you can place the following in the HIPE Editor window.

```
a = Basket()
```

```
a.add("saw")
a.add("hammer")
a.add("chisel")
b = Basket()
b.add("bread")
b.add("cheese")
b.add("milk")
a.print_me()
b.print_me()
```

Now if we hit the "Run all" button then we create two baskets the contents of which will be printed to the command window (bottom left).

This script can be saved using the "File" pulldown menu or save icon (default is ".py" extension).

1.16. Some Useful Extra Items on Scripts

- Some arguments can be optional and can be given a default value. E.g.,

```
def spam(age=32):
  tammy_age = age -- 5
  print -"Tammy is -",tammy_age
  print -"Tammy's brother is -",age
```

Here, spam can be called with zero or one parameter. If no parameters are given it will be called with the default parameter of `age=32`. If a parameter is given with the call then that will be assigned to `age` instead.

Our little script can now be run using, for example,

```
spam()
spam(age=34)
```

- Backquotes (`` ` ``) convert an object to its string representation (so the number 1 can be converted to string "1").

```
age = 32
message = -"Tammy is -"+`age`
print message
```

Here we add (via the plus sign) the string value of `age` to our message.

- The + sign can be used to append string lists.

- One change to make printing easier. We can change to the special method `__str__` so that our last function starts with the line

```
def __str__(self):
```

Instead of

```
def print_me(self):
```

We should also change

```
print -"Basket contains: -" + result
```

to

```
result = -"Basket contains: -" + result
return result
```

Now we can use

```
print a
```

to show our basket contents rather than

```
a.print_me()
```

1.17. Interactivity in Jython Scripts

Sometimes all we need is a script that is launched, performs all its calculations without asking anybody, and then outputs the result and exits. Other times we would like the user to interact, give input while the script is running, take decisions that influence what the script will do. This section takes a look at the tools Jython offers to do just that.

1.17.1. Basic Interactivity

The most common case is for the script to ask the user to input a value. We can use the `raw_input` function, as the tiny example that follows demonstrates.

```
myAnswer = -""
myAnswer = raw_input("Please write something, anything\n")
print -"You wrote -" + myAnswer + -"\nWell done."
```

Here is an interesting fact. When we run this script in HIPE, a small window pops up (see Figure 1.1) with the text we passed to `raw_input`, a box where we can input text and two buttons, OK and Cancel. Save this script and call it `tinyScript.py`, then execute it from the command line, outside HIPE, issuing **python tinyScript.py** or **jython tinyScript.py**, or try double-clicking on the file icon. You will see no fancy windows this time, everything will happen inside a text console. In other words, the window we got is a feature courtesy of HIPE, not a Jython feature.
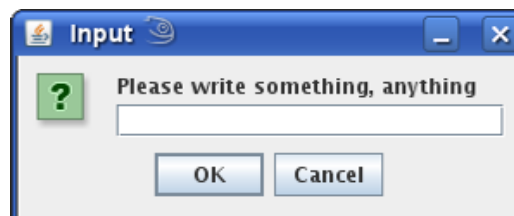
Figure 1.1. The window that appears calling the `raw_input` function from within HIPE.

> **Warning**
>
> Remember that `raw_input` takes everything the user inputs and turns it into a string, including numbers. So be careful when comparing this input to other numbers: you might need to cast your variable to a numerical type.

A fundamental flaw of our little example is that it does not check the input in any way. We could even get away with writing absolutely nothing in the text box, and HIPE would give the seemingly sarcastic reply

```
You wrote
Well done.
```

Of course if we had initialised `myAnswer` to anything else than an empty string, we would get that value in the output. Worse still, if we press the Cancel button, regardless of whether we wrote something or not, the `myAnswer` variable will be set to `None` and the following line will give an error.

One way to have the user input something sensible is to embed the request into a `while` loop, as the following example demonstrates.

```
myAnswer = -""
while myAnswer == -"":
    myAnswer = raw_input("Write something, anything\n")
if myAnswer == None:
    myAnswer = -""
print -"You wrote -" + myAnswer + -"\nWell done."
```

This way the window will not go away until we write something and press OK, and if we try to bypass the check by pressing Cancel the following `if` clause will at least prevent an error on the last line.

More complicated checks can be put in place, for example to make sure that a numerical value stays within the allowed range, and more sophisticated loops may be needed, but the principle is the same.

The above example can also be useful when we want to stop the execution of a script, for whatever reason, and wait before resuming it until the user lets us know that he is in front of the computer and is paying attention. In this case the input does not matter at all, since we just want the user to acknowledge a request by pressing a button.

Well, it works but it is far from optimal. Why having a box for entering text if the text itself does not matter? Wouldn't it be much better to have a window with *Press OK to continue* written on it, the OK button, and nothing else? This is the subject of the next section.

1.17.2. A Little Bit of Swing

To put it simply, *Swing* is the name given to that part of Java that deals with creating graphical user interfaces (or GUIs). Yes, you read correctly: Java, not Jython. Please do not let this scare you. We have used Java bits before, almost without realising it (after all, it is what makes Jython so powerful) and this case will not be different. As a matter of fact, using Swing within Jython is easier than doing so within Java.

This section will teach you enough about Swing to get you started, but if you want to become a GUI guru you may want to look elsewhere. The first chapter of the *Jython Essentials* book has something more to say about Swing. You can find it here:

http://www.oreilly.com/catalog/jythoness/chapter/ch01.html

1.17.2.1. `showMessageDialog`

The first thing we will do is to invoke a Swing method to display a message in a window, together with an OK button:

```
from javax.swing import *
print -"Let's stop for a while"
JOptionPane.showMessageDialog(None, -"Press OK to continue")
print -"Well done."
```

The first line imports the swing package (note that it is `javax` rather than `java`). Then we have the line creating the window, embedded between two lines printing text messages to demonstrate that the script will not advance until we press the OK button.
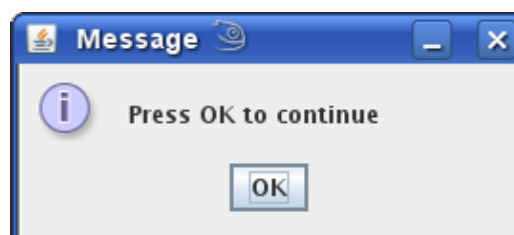
Figure 1.2. The window that appears calling the Swing `showMessageDialog` method.

You have probably noticed that the showMessageDialog method takes two parameters, and we have set the first one to None. It is used to indicate the "parent" element of the dialogue box we are creating. In this case (and in everything that follows) we are just creating a single window and nothing else, so we will not worry about this parameter anymore.

Actually the showMessageDialog can take *more* than two parameters. Notice that the text in the title bar of our window was just "Message". In order to customise it we have to add another parameter, like this:

```
JOptionPane.showMessageDialog(None, -"Press OK to continue", -"Title bar text")
```

Try this and you will get... an error. This is because this third argument *must* go with a fourth one, telling what kind of window we are creating. Let us try again:

```
JOptionPane.showMessageDialog(None, -"Press OK to continue", -"Title bar text", \
JOptionPane.ERROR_MESSAGE)
```
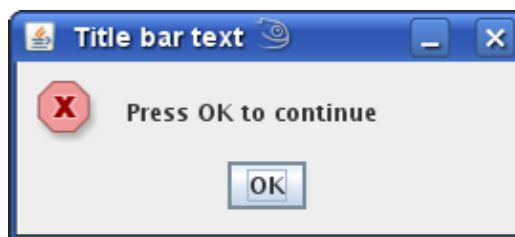

Figure 1.3. Customising the icon and the window title.

Now it works, and it even allows us to change the icon to a nice "error" one. There are a number of possibilities for this fourth parameter, all of which are self-explanatory: ERROR_MESSAGE, INFORMATION_MESSAGE, WARNING_MESSAGE, QUESTION_MESSAGE and PLAIN_MESSAGE. Feel free to try them at your leisure.

If you are sharp-eyed you might have noticed that the previous error message said "expected 2 or 4-5 args; got 3". This mysterious fifth argument is used to add a custom icon to the window, in case you are not satisfied with the predefined ones. Since this is pure eye candy and adds nothing to the functionality of the window, we will not cover it here.

1.17.2.2. `showInputDialog`

Now we would like to take it a step further and create a window for entering text, just like we did with the raw_input function. We just have to use a different method, like this:

```
myAnswer = JOptionPane.showInputDialog(None, -"Please write something, anything")
```
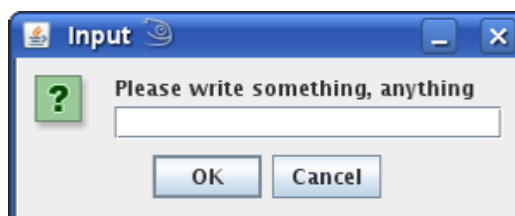

Figure 1.4. The window that appears calling the Swing \showInputDialog method.

You can put this line in the scripts we used to describe the raw_input function and you will obtain the same behaviour, quirks included (even the two windows look exactly the same). The big difference is that, even if you are launching the script from a command line interface *outside* HIPE, a window will still pop up.

Granted, a wealth of additional options is available for this method as well. The ones we saw before are still valid:

```
myAnswer = JOptionPane.showInputDialog(None, -"Please write something, anything", \
"Big question", JOptionPane.QUESTION_MESSAGE)
```

But there is more. We can put a default string of text in the box like this:

```
myAnswer = JOptionPane.showInputDialog(None, -"Please write something, anything", \
"Default text")
```

If we want the user to choose from a predefined set of options, we can use the `showInputDialog` with a whopping seven parameters, as the following script demonstrates:

```
from javax.swing import *
myAnswer = -""
possibleAnswers = ["HIFI", -"PACS", -"SPIRE", -"No clue", -"All three"]
while myAnswer == -"":
  myAnswer = JOptionPane.showInputDialog(None, -"Favourite Herschel instrument?", \
  -"Test", JOptionPane.QUESTION_MESSAGE, None, possibleAnswers, possibleAnswers[4])
if myAnswer == None:
  myAnswer = -""
print -"Your answer is: -" + myAnswer
```
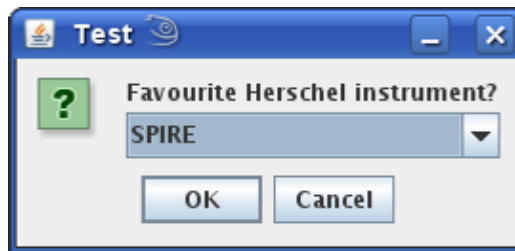

Figure 1.5. A more complex window with a combo box.

Let us go through the parameters one by one:

1. `None`: the "parent" element.

2. `"Favourite Herschel instrument?"`: the window text.

3. `"Test"`: the window title text.

4. `JOptionPane.QUESTION_MESSAGE`: the type of window.

5. `None`: the custom icon. We choose to provide no one and stick with the default one.

6. `possibleAnswers`: the array of possible answers.

7. `possibleAnswers[4]`: the default answer.

1.17.2.3. `showConfirmDialog`

Next we take a look at the `showConfirmDialog` method, which can be used to display a window asking the user to confirm or block a certain action. One example will clarify what we mean:

```
from javax.swing import *
myAnswer = JOptionPane.showConfirmDialog(None, -"Yes or no?")
if myAnswer == 0:              # Now myAnswer is an integer variable
    print -"You agree"
elif myAnswer == 1:
    print -"You disagree"
else:
```

```
    print -"You have no opinion on this"
```
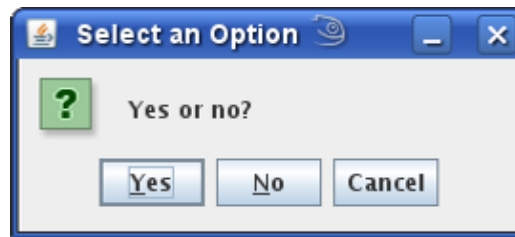

Figure 1.6. Using the Swing `showConfirmDialog` method.

Note that we can use predefined constants to make the code easier to understand, if a little more verbose, as the following, slightly expanded example shows:

```
from javax.swing import *
myAnswer = JOptionPane.showConfirmDialog(None, -"Yes or no?")
if myAnswer == JOptionPane.YES_OPTION:
    print -"You agree"
elif myAnswer == JOptionPane.NO_OPTION:
    print -"You disagree"
elif myAnswer == JOptionPane.CANCEL_OPTION:
    print -"You have no opinion on this"
elif myAnswer == JOptionPane.CLOSED_OPTION:
    print -"You closed the window. How rude!"
```

As always we are free to make things more complicated than that. We can add another two parameters to provide a title for the window and the type of buttons we want:

```
myAnswer = JOptionPane.showConfirmDialog(None, -"Yes or no?", -"Question", \
            JOptionPane.YES_NO_OPTION)
```

Here we decided to drop the Cancel button. Other possible options are YES_NO_CANCEL_OPTION, OK_CANCEL_OPTION, both self-explanatory, and DEFAULT_OPTION, which will just display an OK button.

1.18. Useful Java bits

The Jython language is an implementation of Python written in Java, which means that it is as good-natured yet powerful as Python, but with the added benefit of thousands of packages and classes developed for Java. We will be using some of these classes in the next chapters, and here is a brief description of what they do.

- **The `java.awt` package**. As you already know a *package* is a collection of related classes, like a binder on your desk keeping related documents together. The java.awt package contains all of the classes for painting graphics and images. It is particularly useful for scripts involving plotting and viewing images.

  - **The `java.awt.Color` class**. With this class you can specify a colour for an object. There are thirteen predefined colours available: BLACK, BLUE, CYAN, DARK_GRAY, GRAY, GREEN, LIGHT_GRAY, MAGENTA, ORANGE, PINK, RED, WHITE and YELLOW. If you feel you need a fancier shade you can provide the red, green and blue values individually, as three ints between 0 and 255 or floats between 0.0 and 1.0, like this: java.awt.Color(0.3, 0.2, 0.5). You can also add the alpha (transparency) value as a fourth parameter: 0.0 means completely transparent and 1.0 completely opaque.

  - **The `java.awt.Font` class**. This class allows you to select fonts for annotations on your graphical objects, together with their style and size. The syntax of the constructor (i.e. the special method called to instantiate an object from a class) is like this: Font("SansSerif", 0, 64), where we have the font name, its style code (0 for plain, 1 for bold, 2 for italic) and its size in points.

- **The `java.awt.Window` class**. This class deals with the drawable area of a window on your desktop (not with borders or menu bars). One useful method, especially for plotting, is `setLocation`, inherited from `java.awt.Component`. It accepts two `int` parameters, the `x` and `y` position of the top left corner of the object you want to move.

For more information on these and other classes of the standard Java API you should browse the official Javadoc. If you are looking for a less traumatic introduction to the Java language, the Java Tutorial is an excellent resource.

1.19. Jython and DP Quirks

Every programming language or software system has its *quirks*. Jython and DP are no exception, and this section deals with some of the features you might find confusing.

1.19.1. Two functions for one goal

There are some mathematical function in DP existing in two forms, one in the usual *FirstLetterCapitalised* form (the so-called *CamelCase* convention), the other in *UPPERCASE*. The first form is the recommended way to go, since it is consistent with the rest of the system; the alternative syntax (technically known as *Jython wrapper*) is being kept for backward compatibility, but is not recommended for use in new code and is no longer described in this manual. Examples of Jython wrappers are MATMUL and SOLVE instead of the classes `MatrixMultiply` and `MatrixSolve`, or RESHAPE instead of `Reshape` to change the shape of arrays. You might still bump into them when browsing legacy code.

Unfortunately Jython wrappers are not the only names in uppercase letters, so this is not a good way to identify them, since also e.g. *static instances* (see Section 1.19.3) such as SIN and COS use the same convention.

1.19.2. Long Names versus Short Names

The general rule used in developing the classes used in the DP system is to use long descriptive names, e.g., TableDataset rather than TDset. An exception to the rule is, e.g., IOException rather than InputOutputException

The general rule is that a class name must be self descriptive (easier to remember) which sometimes conflicts with the requirement "I should do every thing by typing three-six letters". The latter was a restriction in F77, and language developers fortunately diverted from that (as it introduced names like CCDF12, CCEFLT, EMPXFF), which are indeed less typing but make the code less (if not completely un-) readable. Exceptions are usually dealing with "well-known" abbreviations. Acronyms such as "IBM Type Writer" is taken to become "IbmTypeWriter" rather than "IndustrialBusinessMachinesTypeWriter."

Any Jython user can create aliases by do things like:

```
TDS=TableDataset
t1=TDS(description="Hello world, this is still a tabledataset!")
print TDS
# herschel.ia.dataset.TableDataset
print t1
# {description="Hello world, this is still a tabledataset!", meta=[], columns=[]}
print t1.__class__
# herschel.ia.dataset.TableDataset
```

Here, in effect, we have created a shortened version of the command we can use to set up a TableDataset called "TDS". We then create a TableDataset, called "t1", which initially contains only a description in the second line. This is equivalent to writing

```
t1=TableDataset(description="Hello world, this is still a tabledataset!")
```

The last two lines indicate the contents of "t1" and the class that created it.

1.19.3. Naming conventions

A potentially confusing aspect to the naming of DP classes is the mix of upper- and lower-case letters. A comprehensive description of the naming conventions used in the HCSS is given in Appendix D and here we just shortly describe the most important aspects.The upper-case/lower-case scheme used in predefined DP classes has the following conventions.

- *Classes*

  Class definitions have names that consist of words of which each first letter is capitalised:

  ```
  MyOwnClass
  TableDataset
  HifiProduct
  ```

- *Class instances -- objects*

  Objects (variables) of a particular class have names that should start with the first letter in lower case. In general, this translates to

  ```
  myOwnClass=MyOwnClass(....)
  table=TableDataset
  a=2
  ```

- *Class instances as constants*

  Certain class instances (or simple variables) are used as constants. The convention is to use names with all their letters capitalised and words separated by an underscore '_'. These are sometimes referred to as static instances. An example is SIN: it is the only (allowed) instance of class Sin, as it does not make sense to have multiple instances of these. Examples are:

  ```
  VARIANCE
  IS_FINITE
  ALL_PRESENT
  ```

1.19.4. Miscellaneous quirks

- **Working directories.** Restrictions are placed on dealing with working directories due to the use of Java. This is discussed in Section B.6.

- **Loops, indentation and blank line usage.** Indentation in loops is very strict within HIPE. Blank lines can have particular significance, particularly with respect to setting up loops. These quirks are described in Section B.8.

- **Logical operators.** The presence of Jython original features together with DP specific ones can result in counter-intuitive behaviour and unexpected results Section 3.7 in Chapter 3 deals with these quirks.

- **Incompatible numeric types.** Jython has its own primitive numeric types, but Java numeric types can be used as well. Mixing Java and Jython types (and even using Java types on their own) can lead to strange errors that are explained in Section 1.5.1.1.

- **Script length.** Each Jython script is compiled by the Java virtual machine into a single non-native, non-abstract method and such Java methods cannot exceed certain limit, usually 65536 bytes. If your Jython script is very long (more than a few thousands lines) then it is advisable to split it into separate scripts.

Chapter 2. Arrays, datasets and products

2.1. Introduction

This chapter aims to familiarize you with the DP Array data objects, Datasets and Algorithms concepts. This is not an exhaustive reference to all the functionality provided, the full set of available array object and dataset capabilities are discussed in the *herschel.ia.numeric* and *herschel.ia.dataset* packages Javadoc.

There are three types of basic datasets:

- array datasets (datasets containing single `ArrayData` objects, holding numbers, strings, etc. in 1D, 2D, 3D, 4D or 5D)

- table datasets (x rows by y columns of numeric or string arrays). Table datasets can have columns of various data types mixed in the same dataset and can also contain unit and descriptive information for individual columns.

- composite datasets (combines multiple connected arrays/tables in a single dataset).

One of the major advantages of DP numeric array objects (as opposed to Jython lists) is the ability to do array arithmetic in single line commands rather than having to loop through arrays.

In this chapter, we discuss how to formulate and use each array object and dataset type.

2.2. Getting started

All classes and methods associated with handling datasets and numeric functions are automatically loaded when the DP session is started in this manner.

The DP *numeric* package currently contains many functions and is discussed in more detail in Chapter 3. Here we include the use of portions of it to help illustrate how datasets may be handled.

2.3. Types of Array Data Objects

DP numeric array data objects can have up to 5 dimensions and have the types shown in the following table.

Table 2.1. Numeric types available in DP (N = 1...5)

Name	Type	Dimensions		
		1	2	3+
BoolNd	boolean	yes	yes	yes
ByteNd	byte	yes	yes	yes
ShortNd	short	yes	yes	yes
IntNd	integer	yes	yes	yes
LongNd	long	yes	yes	yes
FloatNd	float	yes	yes	yes
DoubleNd	double	yes	yes	yes
ComplexNd	complex	yes	yes	yes

Name	Type	Dimensions		
String1d ❶	string	yes	NO	NO

❶   The `String1d` array type is not strictly numeric.

2.3.1. DP Numeric Array Access and Slicing

The numeric package introduces the following square brackets notation:

```
[i_0,...,i_n-1]
```

where each element is separated by a comma, and the number of elements must be equal to the rank of the array. Arrays are zero-based which means the first element of an array has index 0 (zero) and the index of the last element of an array is `array.length()-1`.

In addition the package supports the colon (`:`) notation to designate a slice. A slice is a range of indices defined as `i:j`, where `i` is the starting index and inclusive, and it is zero if not specified. The ending index `j` is exclusive and it is equal to `array.length()` if not specified and `array.length()-j` if negative.

The following example illustrates the access to elements in a multi-dimensional array and the use of slices. More examples can be found in the section on Multi-Dimensional Arrays.

```
# define something that is like a rectangular 2x3 array:
#   1 2 3
#   4 5 6
x=Int2d([[1,2,3],[4,5,6]])# Int1d can swallow the jython sequence.
print x                    # [[1,2,3],[4,5,6]]
print x[1]                 # 2 (second element of the first row)
print x[1,:]               # access a row i.e. [4,5,6]
print x[1,1]               # access an individual element i.e. 5
print x[:,:]               # [[1,2,3],[4,5,6]]
print x[:,1]               # access a column i.e. [2,5]
```

2.4. Creating a Simple 1D DP Numeric Array

In order to create an array data object we only need to do something like the following:

```
a = Int1d()
```

This provides us with an empty integer array. We can now add elements to this by

```
a.append(2)
```

Or

```
a.append(Int1d([1,2,3,4,5]))
```

to append a whole 1D integer array.

Alternately, we could have created the array in one go, like this:

```
a = Int1d([1,2,3,4,5])
```

The following show various ways in which numeric 1D arrays can be created in the DP environment.

```
y = Double1d([1.0,2.0,3.0,4.0]) # Create from a Jython array
y = Double1d(4) # [0.0,0.0,0.0,0.0]
y = Double1d(4, 42.0) # [42.0,42.0,42.0,42.0]
```

```
y = Double1d.range(4) # [0.0,1.0,2.0,3.0]
```

2.5. Creating and Handling Complex Array Data Objects

The numeric library has a `Complex` class and a `ComplexNd` class for N-dimensional arrays of complex numbers (N = 1, 2, 3, 4 or 5).

```
z = Complex1d([1,2,3,4],[4,3,2,1]) # Set up complex array
print z # [(1.0+4.0j),(2.0+3.0j),(3.0+2.0j),(4.0+1.0j)]
print z.getReal() # Print real part
print z.getImag() # Print imaginary part
print z.conjugate() # [(1.0-4.0j),(2.0-3.0j),(3.0-2.0j),(4.0-1.0j)]
```

Complex numbers in the numeric package are constructed using the `Complex` constructor (with an upper-case 'C'):

```
z1 = 2 + 3j # Jython complex (2+3j)
z2 = Complex(2,3) # Numeric Complex (2.0+3.0j)
```

In other respects, `Complex` arrays are used in much the same way as `Double` arrays. Their main use, at present, is for discrete Fourier transforms.

2.6. Creating and Accessing Multi-Dimensional Array Data Objects

Creating and manipulating multi-dimensional arrays occurs in a similar way to the 1D case. The DP numeric library supports arrays of up to 5 dimensions. For example, to create a `Double2d` array:

```
x = Double2d([[2,4,6],[1,3,5]])
```

Multi-dimensional arrays are conceptually arrays of lower-dimensional arrays. For a two-dimensional array, the first subscript selects a row and the second subscript selects an element within that row (the column).

Note

This is the opposite order to some other computer languages, but it is the same behaviour as in the Java programming language.

For example:

```
print x[1,:] # Get row 1 i.e. [1.0,3.0,5.0]
print x[1,2] # 5.0, the element in row 1, column 2
```

Note: indexing multi-dimensional arrays is done differently in DP numeric arrays as compared to Jython arrays. The following code examples show the syntax for Jython and DP numeric arrays. The reason for this is to allow slicing on multi-dimensional arrays in DP which is technically not possible using the Jython syntax.

```
# Jython array:
x = [[1,2,3,4],[5,6,7,8]]
print x[1][2]      # 7
print x[1][1:3]    # 6, 7
```

```
# DP numeric array:
y = Int2d([[1,2,3,4],[5,6,7,8]])
print y[1,2]       # 7
```

```
print y[1,1:3]      # 6, 7
```

Individual elements or slices can be set as follows:

```
x[1,2] = 22 # Set an element in place
x[0,1:3] = 42
print x # [
        # [2.0,42.0,42.0],
        # [1.0,3.0,22.0]
        # -]
```

It is possible to set a row to a copy of a 1d array of the same length:

```
x[0,:] = [5,6,7,8]        # Set a row to (a copy of) a Jython array
y[1,:] = Int1d([9,7,6,5]) # Set a row to a Double1d array
```

2.6.1. A note on array ordering

Look again at the first example of [Section 2.6](#):

```
x = Double2d([[2,4,6],[1,3,5]])
```

This line of code creates an array of two rows and three columns. The element corresponding to the *i*-th row and *j*-th column can be accessed like this:

```
x[i, j]
```

The values are stored sequentially in memory as follows:

```
[2 4 6 1 3 5]
```

This means that, if we go through the array elements as they are stored in memory, their indices would vary as follows:

```
x[0,0] x[0,1] x[0,2] x[1,0] x[1,1] x[1,2]
```

That is, index *j* varies *more rapidly* than index *i*. We can generalise to more than two dimensions by saying that *the rightmost index varies most rapidly*. This is called *row-major* ordering, and is the convention followed by languages such as Java and C, *but not Fortran*.

This has an implication on performance. When looping through a multidimensional array, it is more efficient to read its elements in the order they are stored in memory.

Confusion may also arise when dealing with images, which are stored as two-dimensional arrays. If we visualize the array with horizontal rows and vertical columns, then the number of rows and columns represents the size of the vertical (y) and horizontal (x) side of the image, respectively. When accessing a particular pixel (array element), you have to specify the *y* coordinate *before* the *x* coordinate:

```
myImage(y, x)
```

2.7. Adding Attributes to Create an Array Dataset

Let's start by creating a simple dataset. Let's assume that we want to create a dataset containing one component: a 1D array of double precision numbers (doubles in an array we will call 'x').

Type in the following steps (without the comments preceded by '#'):

```
x = Double1d.range(10)  # ❶
```

```
s = ArrayDataset(data=x,description="range of double values")  # ❷
```

❶   The `range()` function creates a 1D array of integers with the values 0, 1, 2...9. Putting `Double1d` in the front converts the array values to doubles.

❷   This actually creates the array dataset with data being the array x of values 0.0, 1.0, 2.0...9.0 and some associated information, a description.

This creates an object x, corresponding to a 1D array of 10 doubles from 0 to 9, and writes that to a dataset object, s, which also contains a description of the dataset. The `range` command produces ten integer numbers from 0 to 9. This is placed in a 1D array of doubles by the first line.

Now let's look at the contents of the dataset s:

```
print s
```

If you want to be specific and print individual components of the dataset, you may do so using the special description and data attributes:

```
print s.description # Just print the description that is attached to the dataset
print s.data        # Print only the data contained in the dataset
```

And even individual elements of the data component:

```
print s.data[2] # View the value of the third element of the array
                # contained in the dataset
```

2.7.1. Dataset Attributes and Metadata

In the previous section, we have seen that the ArrayDataset s possesses at least 2 attributes: `description` and `data`. They have in addition a third attribute not so far illustrated, `meta`. The `description` and `meta` attributes are common across all dataset types.

The `description` attribute is used to store a human-readable text that helps the user to understand the role of the dataset.

The `meta` attribute stores a map of keyword-value pairs of data that can be used to identify that data in a database (for example) - the so-called *meta-data*. Examples of metadata for an observation include *the date of the current observation; the name of the source; the coordinates of the source*, etc. **These are basically the DP equivalent of FITS keywords**. The allowed data types for meta-data elements are String, Double, Boolean, Long, and Date (e.g., `StringParameter`, `DoubleParameter` etc.). See the JavaDoc for the `MetaData` class for more information on the allowed types.

The following code snippet shows how to add parameter information (in the form of strings or doubles) to the `meta` attribute:

```
s.meta["observation"] = StringParameter("NGC 4151")
s.meta["principal investigator"] = StringParameter("Anthony Marston")
s.meta["ra"] = DoubleParameter(182.836)
s.meta["dec"] = DoubleParameter(39.405)
```

These are actually shortcuts to Java usage. For example, the first line could also have been written as

```
s.getMeta().set("observation", StringParameter("NGC4151"))
```

2.8. Creating and Viewing a TableDataset

What is often required is to store data in a tabular format with N columns. The `TableDataset` provides such a means. A `TableDataset` is made up of a number of columns. Each column contains

an `ArrayDataset` (data), a description and a quantity (unit -- require the Unit package import, see below) value associated with the `ArrayDataset`. Each `ArrayDataset` can have up to 5 dimensions and can be of varying types. In the following example, a `TableDataset` is created with 3 columns each containing a 1D dataset, one being a sequence of numbers from 1 to 100, the second being the sine value of each of the numbers in the first column, and the final column containing the values in the first column multiplied by 100. The column names are `x`, `sin` and `y` respectively.

Note

For reasons of flexibility, memory consumption and performance, this class is not checking whether all columns are of the same length: this is the responsibility of the user.

```
from herschel.share.unit import *  # to allow the use of the Unit package

x = Double1d.range(100)
t = TableDataset(description="This is a table")  # ❶
t["x"] = Column(data=x, unit=Duration.SECONDS)  # ❷
t["sin"] = Column(data=SIN(x),description="sin(x)")  # ❸
t["y"] = Column(data=x*100,description="x*100")
```

❶   This sets up the table dataset with an associated description
❷   This creates our first column which has the data, `x` and its associated units, which in this case is a time duration of SECONDS.
❸   Here we have applied the `SIN` function from the numeric package, and we have also added a description for the second column.

Tabledatasets can be viewed using the DatasetInspector GUI button. Values can also be obtained using the following steps which show how the data can be listed:

```
print t            # Print a TableDataset called t (see above)
print t.meta       # Print the metadata (empty in this case)
print t["x"]       # Print a column by name
print t[2]         # Print a column by index
print t[2].data    # Print the data inside the column
a = t[2].data      # Assign data in column to a list variable, -"a".
print t[2].data[4] # Print element with index=4 in the last (third!) column
b = t[2].data[4]   # Assign the data value to variable -"b".
print t[2].description # Prints column description only
print t["x"].unit  # print the associated unit values for the column
```

Alternately, we can access columns via the getColumn method

```
print t.getColumn("y") # Print a column by name
print t.getColumn(2) # Print a column by index
print t.getColumn(2).data # Print the data inside the column
print t.getColumn(2).data[4] # Print element with index=4 in the third column
print t.getColumn(2).description # Prints column description only
```

We can also get row values

```
print t.getRow(1)  # Gets a list of the values in the second row.
```

And here is how data can be modified:

```
print t["y"].data[0]
t["y"].data[0]=999.
print t["y"].data[0]
```

We may also get and set values at a position in a TableDataset.

```
t.getValueAt(0,1) # gets the value contained in row=0, column=1
t.setValueAt(30.5, 0, 1) # sets the value 30.5 at row=0, column=1
```

2.8.1. Row-wise appending of TableDatasets

It is possible to append the data from one table dataset to data in another, provided that they have the same number of columns and each column in either dataset is of the same type. The following example adds `t2` as a row to table `t1`.

```
t1 = TableDataset()
t1["x"] = Column(data=Int1d.range(5))
t1["y"] = Column(data=Double1d.range(5))
t2 = TableDataset()
t2["a"] = Column(data=Int1d.range(10))
t2["b"] = Column(data=Double1d.range(10))

# The following will append the data in t2 to the data in t1
# t1.rowCount will then report 15 rows:
t1.addRow(t2)
```

If we now use `print  t1["x"].data` we can see that the "x" column has the values [0,1,2,3,4,0,1,2,3,4,5,6,7,8,9].

2.8.2. Assigning Units

This section exaplins what units can be assigned and how they may be manipulated. As we have noted above, we can assign units to the columns in our dataset. in order to use the Unit package we have to import it:

```
from herschel.share.unit import *
```

Note that the Unit package are used in the whole HCSS and not only in the interactive analysis, that is why it is part of the herschel.share library.

The units fall into several category types, as they are shown in alphabetical order in Table 2.2. To assign a unit the type and value s required to be given. For example -- the variable "a" can be assigned to be a unit of angle in degrees with

```
a = Angle.DEGREES  # Type.VALUE
```

This can be associated with a column's unit in a table using

```
t["x"].unit = Angle.DEGREES
```

Table 2.2. All available basic units types

Type	VALUES
Acceleration	METERS_PER_SECOND_SQUARED
Angle	RADIANS, DEGREES, MINUTES_ARC, SECONDS_ARC
AngularMomentum	JOULE_SECOND
AngularSpeed	RADIANS_PER_SECOND, DEGREES_PER_SECOND
Area	SQUARE_METERS, SQUARE_KILOMETERS
Constant	H_PLANCK, K_BOLTZMANN, ELECTRON_CHARGE, SPEED_OF_LIGHT
Duration	SECONDS, MINUTES, HOURS, DAYS
ElectricCapacitance	FARADS, MILLIFARADS, MICROFARADS, NANOFARADS, PICOFARADS
ElectricCharge	COULOMBS

Type	VALUES
ElectricConductance	SIEMENS
ElectricCurrent	AMPERES, MILLIAMPERES
ElectricInductance	HENRIES
ElectricPotential	VOLTS, MILLIVOLTS
ElectricResistance	OHMS
Energy	JOULES, ERGS, ELECTRON_VOLTS
Entropy	JOULES_PER_KELVIN
Flux density	JOULES_PER_SQUARE_METER, JANSKYS, MILLIJANSKYS, MICROJANSKYS
Force	NEWTONS, DYNES
Frequency	HERTZ, KILOHERTZ, MEGAHERTZ, GIGAHERTZ, TERAHERTZ
Length	METERS, ANGSTROMS, KILOMETERS, CENTIMETERS, MILLIMETERS, MICROMETERS
Mass	GRAMS, KILOGRAMS
NEP (Noise Equivalent Power)	WATTS_PER_SQRT_HERTZ
Power	WATTS, KILOWATTS, MEGAWATTS
Pressure	PASCALS, BARS, MILLIBARS
Scalar	This class represents scalar units and provides some constants:ONE, PERCENT,DECIBELS
SolidAngle	STERADIANS, SQUARE_MINUTES_ARC, SQUARE_SECONDS_ARC
Speed	KILOMETERS_PER_SECOND, METERS_PER_SECOND
Temperature	CELSIUS, KELVIN
ThermalConductivity	WATTS_PER_METER_KELVIN
TimeInstant	TAI, UTC
WaveNumber	RECIPROCAL_METERS, RECIPROCAL_CENTIMETERS

2.8.2.1. Manipulating Units

We may manipulate units to obtain derived units. Examples are the following

```
N  = Force.NEWTONS
m  = Length.METERS
m2 = m**2              # Square meters
Pa = N -/ m2            # Pascals
J  = N * m             # Joules
```

2.8.2.2. Converting Units to Strings and Back Again

We can convert a unit variable to a string in several ways:

```
A = Length.ANGSTROMS
print A               # angstrom [1.0E-10 m], no conversion
print A.name          # angstrom. This is a string quantity.
print A.dialogName    # Angstrom symbol. This is a string quantity.
um = Length.MICROMETERS
print um              # micrometer [1.0E-6 m], no conversion, includes factor
                      # with respect to SI unit
print um.name         # micrometer, only ASCII characters. This is a string.
print um.dialogName   # μm. This is a string quantity.
```

We can also convert a string to a unit

```
print Unit.parse("km s-1")
# or print (Unit.parse("km") -/ Unit.parse("s"))
print Unit.parse("km s-1") # Speed.KILOMETERS_PER_SECOND
print Unit.parse("arcsec") # Angle.SECONDS_ARC)
print Unit.parse("eV") # Energy.ELECTRON_VOLTS)
print Unit.parse("cm") # Length.CENTIMETERS)
print Unit.parse("mm") # Length.MILLIMETERS)
print Unit.parse("microm") # Length.MICROMETERS)
```

2.8.2.3. Derived Units

We can also provide derived units by application of .milli, .micro and .nano methods.

```
s  = Duration.SECONDS
us = s.micro  # micro seconds
ns = s.nano   # nano seconds
```

2.8.2.4. Conversion to SI and Other Units

If the SI unit is needed rather than the unit used then SI unit and the factor between the two can be provided.

```
print Angle.DEGREES.asSI  # gives unit as Angle.RADIANS
print Energy.ERGS.asSI    # gives unit as Energy.JOULES
print Speed.KILOMETERS_PER_HOUR.asSI # gives unit as Speed.METERS_PER_SECOND
print Unit.parse("g cm s-2").asSI # gives unit as Unit.parse("kg m s-2")
#
print Length.ANGSTROMS.toSI        # 1.0E-10
print Duration.HOURS.toSI          # 3600.0
print FluxDensity.MILLIJANSKYS.toSI  # 1.0E-29
print Unit.parse("g cm s-2").toSI   # 1.0E-5
# or factor compared to other units
min = Duration.MINUTES
ms  = Duration.MILLISECONDS
print min.to(ms)        # 60000.0
mV = Unit.parse("mV")  # millivolts
print mV.to(mV.asSI)   # 0.001; same as mV.toSI
```

2.8.2.5. Physical Constants

Physical constants can also be provided to the system with their correct units, e.g.

```
h = Constant.H_PLANCK
print h.value  # 6.62606896E-34
print h.unit   # J s
print h        # 6.62606896E-34 J s
k = Constant.K_BOLTZMANN
print k.value  # 1.3806505E-23
print k.unit   # J K-1
print k        # 1.3806505E-23 J K-1
```

2.8.2.6. Unit Compatibility

We can compare units to see if they are of compatible types.

```
kg = Mass.KILOGRAMS
g  = Mass.GRAMS
m  = Length.METERS
print kg.isCompatible(g)                       # true
print kg.isCompatible(m)                       # false
print kg.isCompatible(Mass)                    # true
print kg.isCompatible(Area)                    # false
```

```
print Unit.parse("g cm s-2").isCompatible(Force)   # true
print Unit.parse("g cm s-2").isCompatible(Power)   # false
```

2.8.2.7. Unit Equivalence

We can use the `.isEquivalent` method to determine if two unit types are the same.

```
kg  = Mass.KILOGRAMS
s   = Duration.SECONDS
m   = Length.METERS
N   = Force.NEWTONS
dyn = Force.DYNES
print N.isEquivalent(dyn)            # false
print N.isEquivalent(kg * m -/ s**2)  # true
```

2.9. Creating and Accessing a Composite Dataset

The `ArrayDataset` and `TableDataset` types enable the user to encapsulate arrays and tables of primitive data types easily. However, they do not allow arbitrary structures of data, or data within data, to be constructed. Examples of complex datasets are grouped observations (making a map with an offset reference position, for instance), which could have 1D and 2D array data together with a table which might contain (for example) calibration data. Such complex structures can be built using the `CompositeDataset`. Example 2.1 creates a `CompositeDataset` containing in turn an `ArrayDataset`, a `TableDataset`, a few `StringParameters`, and another nested `CompositeDataset`. It also illustrates how we can access the components of the composite dataset.

```
# First we set up a one-dimensional array of doubles (0.0, 1.0 -... 9.0)
x = Double1d.range(10)
# Then we create an array dataset with an added description
s = ArrayDataset(data=x,description="Range of doubles")
# This sets up an empty table with a description
t = TableDataset(description="This is a table")
# The array -'x' is then added to the table and given a
# column heading -"x"
t["x"]=Column(x)
# Each of the array elements of -'x' is multiplied by 4
# and becomes the data in the table column labeled -"y".
# The table column also has a description added to it.
t["y"]=Column(data=x*4,description="x*4")
# c is an empty composite dataset.
c=CompositeDataset()
# We add a description to c
c.description="This is a composite dataset. It contains three datasets!"
# We add the author's name as a string parameter
c.meta["author"]=StringParameter("Jorgo Bakker")
# We input a version number as a string parameter
c.meta["version"]=StringParameter("2.0")
# We put the array dataset s into the composite dataset c
# and give it the name mySimple so that we can refer to it
c["mySimple"] = s
# We do the same for the table
c["myTable"] = t
# This just shows you can add a composite dataset into another
# composite dataset (nesting)
c["myNest"] = CompositeDataset("Empty nested composite dataset")

print c             # View contents of the complex dataset.
tab = c["myTable"]  # Gets our TableDataset back. Now called -"tab".
print tab           # We see that it has two columns called -"x" and -"y"
print tab["x"]      # Prints out what is in the -"x" column.
print tab["x"].data # To just print out the data values.
```

Example 2.1. Example of how to create a composite dataset

2.10. Spectrum Datasets

Spectra are contained within datasets that also contain raw data counts together with metadata that allows for the correct handling of combinations of spectra (e.g., spectral arithmetic) and display of spectra. Basic spectral types are `SpectralSegment`, `Spectrum1d` and `Spectrum2d`.

2.10.1. Spectrum1d and SpectralSegments

A one-dimensional representation of a spectrum. Container has a TableDataset() that has columns for flux, flag, weights and numbered segments (components of the 1d spectrum). It contains

- A flux column (Double1d). This can be obtained from a SpectralSegment using the getFlux() method. For example; a = %spectrum1d_name%.getFlux().

- A wavelength/frequency column (Double1d). The wavelength column can be obtained using the getWave() method.

- A weight column (Double1d). The weight column can be obtained using the getWeight() method.

- A segments column (Double1d). The segments column can be obtained using the getSegment() method.

- A flag column (Int1d). The flags can be obtained using the getFlag() method.

A Spectrum1d can also have metadata (header information) added. The following illustrates how a Spectrum1d dataset can be built from scratch.

```
flux = Double1d([12.2,12.5,13.0,11.8,11.9,12.6,14.2,15.8,12.2,15.2])
segs = Int1d([0,0,0,0,0,1,1,1,1,1])  # segment id for each point
wave = Double1d([1000.0,1000.2,1000.4,1000.6,1000.78,
    \ 1100.0,1100.2,1100.4,1100.6,1100.78])
flag = Int1d(10) + 1
weight = Int1d(10) + 1.0
a = Spectrum1d(flux,weight,flag,segs) #indicate the fluxes and segments.
a.set("wave", wave) # add the wavelengths column
a.setMeta("name","Arp220") # sets keyword name in metadata of Spectrum
# other metadata can be added, as needed.
print a.getWave() # shows the -"wave" column
# Using the Dataset viewer, the full information can be viewed
```

The spectrum can be made of several segments. A `SpectralSegment` is the smallest spectrum component dealt with by the DP system. This can be a piece of a spectrum extracted from a larger one-dimensional spectrum to be used for fitting purposes (for example). It can be extracted from a Spectrum1d using the following.

```
b=a.getSpectralSegment(1) # get second spectral segment (numbering starts at 0)
print b.getWave() # provides the wavelengths associated with this segment
```

Many of the spectral tools (arithmetic, fitters) work with the basic unit of a spectral segment.

2.10.2. Spectrum2d

For multiple spectra taken in an observation, a 2D structure is required. The components of a Spectrum2d dataset is similar to that of a Spectrum1d dataset, except for having a second dimension. An additional component is the ability to contain subbands. A clear example of the usefulness of this comes in the output from the HIFI spectrometers where several CCD or autocorrelator readouts lead to several "chunks" (subbands) of spectra in one data frame. Having subbands is an option for the Specrum2d. It contains

- A flux column (Double2d). This can be obtained from a SpectralSegment using the getFlux() method. For example; a = %spectrum1d_name%.getFlux().

- A wavelength/frequency column (Double2d). The wavelength column can be obtained using the getWave() method.

- A weight column (Double2d). The weight column can be obtained using the getWeight() method.

- A flag column (Int2d). The flags can be obtained using the getFlag() method.

- (optional) a subbandstart column (Int1d). Indicates where in the arrays that a subband starts.

- (optional) a subbandlength column (Int1d). Indicates the length of array section that a subband takes up.

The number of channels is automatically generated in the metadata when setting up a Spectrum2d. An example of setting up a Spectrum2d from scratch is given below.

```
flux2 = Double2d([[12.2,12.5,13.6,12.8],[12.8,12.2,13.3,12.9],
\ [10.2,14.5,12.5,11.4],[12.2,12.5,13.6,12.8]])
flag2 = Int2d([[1,1,1,1],[1,1,1,1],[1,1,1,1],[1,1,1,1]])
weight2 = Double2d([[1,1,1,1],[1,1,1,1],[1,1,1,1],[1,1,1,1]])
a2 = Spectrum2d(flux2,weight2,flag2) # sets up 4 channels each with 4 pixels
wave2 = Double2d([[1000.0,1000.2,1000.4,1000.6],[1000.0,1000.2,1000.4,1000.6],
  \ [1000.0,1000.2,1000.4,1000.6],[1000.0,1000.2,1000.4,1000.6]])
a2.set("wave", wave2) # add the wavelengths
print a2.getWave()  # to print out the wavelengths
print a2.getFlux()  # to print out the fluxes.
```

We can also set up a Spectrum2d with associated subbands. This basically allows us to set up, in one dataset, a container which holds many individual spectra which as many subbands each covering a different wavelength range, if necessary (e.g., with the individual subbands of the HRS spectrometer of HIFI). This forms the basis of how spectral observations, which typically are made up of many frames, are stored in the Herschel DP environment.

```
# Now deal with subbands.
# Create the container for the spectra
a3 = Spectrum2d()
# indicate the number of subbands it will have
a3.setSubbands(2)
a3.setSubbandStart(Int1d([0,2]))
a3.setSubbandLength(Int1d([2,2]))
flux3 = Double2d([[12.2,12.5,13.6,12.8],[12.8,12.2,13.3,12.9]])
flux4 = Double2d([[10.2,14.5,12.5,11.4],[12.2,12.5,13.6,12.8]])
a3.set("flux_1",flux3)
a3.set("flux_2",flux4)
print a3.getFlux(1)
wave3 = Double2d([[1000.0,1000.2,1000.4,1000.6],[1000.0,1000.2,1000.4,1000.6]])
a3.set("wave_1",wave3)
a3.set("wave_2",wave3)
#get wavelengths for second subband
# note that there are two sets of measurements
print a3.getWave(2)
#get fluxes for first set of measurements
# of subband number 1.
print a3.getFlux(1).get(0)
# or second set
print a3.getFlux(1).get(1)
# this way you can go through multiple
# measurements using the same subband that are
# stored in the same dataset.
# We can do the same for wavelengths, e.g.,
print a3.getWave(1).get(0)
# instrument pipelines producing spectra store the data in Spectrum2d
# or a variant (see next section).
```

2.10.3. Expanding Spectrum1d and Spectrum2d Datasets

Extensions to the basic Spectrum1d and Spectrum2d datasets have been created that allow for more convenient access to specific instrument data types. Typically, the full spectral information, including metadata, is created from the original instrument dataframes and housekeeping information coming from the spacecraft. However, it can be instructive to formulate things from their basic components.

2.10.3.1. HIFI Extensions

Examples of HIFI extensions to the Spectrum1d and Spectrum2d datasets are the `WbsSpectrumDataset` and `HrsSpectrumDataset` available for the two types of spectrometer data from HIFI. These can be created by obtaining HIFI dataframes and housekeeping telemetry source packets (these are not generally available to most users).

```
# creating a WBS spectrum dataset
from herschel.hifi.pipeline.product import *
w = WbsSpectrumDataset(array of WBS dataframes, array of HK telemetry)
```

Such a spectrum dataset automatically includes more metadata such as observation identification and data creation date. It can also contain the information for the wavelength as a model -- typically polynomial fit information.

Displaying the table of dataset, for each spectrum not only is flux and wavelength listed but other, HIFI-specific, information such as chopper position and on-board buffer storing the data (see Fig.***).

Typical observations actually contain groupings of such datasets. For example, internal flux calibrator dataframes, science dataframes and frequency calibrator data frames. These are typically grouped together in a HIFI timeline product. So a typical HIFI observation with all four spectrometers used would have four HIFI timeline products.

```
# Creating a HIFI timeline product
from herschel.hifi.pipeline.product import *
htp = HifiTimelineProduct(array of WBS dataframes, array of HK telemetry)
```

For the most part users will not need to create the datasets/products but will need to access the data in them. We can use the `getFlux()` and `getWave()` methods as before. For HIFI spectra, the `getWave()` method provides the IF frequency values. The lower or upper sideband frequencies can also be obtained using the `getLsbFrequency()` or `getUsbFrequency()` methods. So we can crudely plot -- with labels to be attached later -- the spectrum (upper or lower sideband) using the following.

```
# Continuing from above.
# Get the first dataset in the product
wbs = htp.get(1)
# Plot of flux against IF frequency
p = PlotXY(wbs.getWave().get(1),wbs.getFlux().get(1))
# This provides a plot of the second frame, called frame number 1.
# Similar but now will plot the LSB frequency which takes
# the local oscillator frequency information into account
p = PlotXY(wbs.getLsbFrequency().get(1),wbs.getFlux().get(1))
```

2.10.3.2. SPIRE extensions to Spectrum1d

The SPIRE instrument also uses an extension of `Spectrum1d`. The basic component dataset for the spectrum obtained by a single SPIRE pixel is the `SpireSpectrum1d`. As opposed to `Spectrum1d`, complex data are possible (stores Numeric1d inputs as Complex1d). The data is

composed of complex values of flux and flux error with associated units. A mask can also be added (type `Int1d`).

Individual spectra from separate pixels can be grouped together to formulate a single SPIRE scan dataset. This in turn can be grouped into a set of scans that would be more typical of a single SPIRE observation.

```
from herschel.share.unit import *
from herschel.spire.ia.dataset import *
c = Complex1d([2+3j, 3+2.1j,3.6 +2.4j,0.9+2.1j])
err = Complex1d([0.2+0.2j, 0.8+0.3j,0.4+0.3j,0.15+0.1j])
flu = FluxDensity.JANSKYS
wu = WaveNumber.RECIPROCAL_METER
wn = Double1d([0.3,0.4,0.5,0.6])
mask = Int1d([1,1,1,1])
sps = SpireSpectrum1d("Pixel name")
sps.setComplexFlux(c,flu)
sps.setComplexFluxError(err,flu)
sps.setWavenumber(wn,wu)
sps.setMask(mask)
# Now we can get the data by replacing set by get,
# and removing the arguments, e.g.,
sps.getComplexFlux() # returns the flux data
# and we can get the units separately, e.g.,
sps.getComplexFluxUnits()
## Now we can place a number of pixels in a single unit
## a SpireSpectrumCompositeDataset.
## Create sps, sps1, sps2, sps3 etc.
spire_cds = SpireSpectrumCompositeDataset("Scan number")
## Scan number can be a string name (as above) or a long numeric value.
## add pixels of data.....
spire_cds.setPixel(sps)
spire_cds.setPixel(sps1)
spire_cds.setPixel(sps2)
spire_cds.setPixel(sps3)
## pixel names are as set up in the original SpireSpectrum1d
## we can get a pixel using
wanted_sps = spire_cds.getPixel("Pixel name")
### Most SPIRE spectrometer observations are composed of many scans
### which we can then place several composite datasets in a single dataset.
spire_sds =SpectrometerDetectorSpectrum() # create empty dataset
spire_sds.setScan(spire_cds) # add in scan, given next scan number available = 0.
spire_sds.setScan(spire_cds1) # add in scan, given next scan number available = 1.
### Now access a scan.
wanted_cds = spire_sds.getScan(0) # for the first scan
```

2.10.3.3. PACS Spectrum1d and Spectrum2d extensions

PACS spectral is based on handling the Frames and Ramps based on the readout of the PACS spectrometer. The handling of these data is currently discussed in the PCSS User's Manual.

2.11. Image and cube datasets

Image and cube datasets are made of `Double2d` and `Double3d` components representing intensity, masks and errors, in addition to metadata providing coordinate information.

A `SimpleImage` is a standard two-dimensional image represented by a `Numeric2d` (such as `Double2d` or `Int2d`). The following components can be added:

- The `Error`, as a `Numeric2d`.

- The `Exposure`, as a `Numeric2d`.

- A set of flags as a `Flag` object.

- Measurement units as a `Unit` object.

- WCS information as a `Wcs` object.

An example of creating a `SimpleImage` from an imported JPG file is given below. You can find the `ngc6992.jpg` file in the `/data/ia/demo/data` folder of your HIPE installation.

```
from herschel.share.unit import *
# Choose units
myQuant = FluxDensity.MILLIJANSKYS
# Create WCS
myWcs = Wcs(crpix1 = 29, crpix2 = 29, crval1 = 30.0, crval2 = --22.5)
# Create the simple image with an assigned WCS and a description
myImage = SimpleImage(description="Veil nebula", unit = myQuant, wcs = myWcs)
# Import an image
# Note: we assume that the current directory is -"bin"
# in your HIPE installation.
importImage(myImage, -"../data/ia/demo/data/ngc6992.jpg")
# Assign a reference wavelength to the image
myImage.setWavelength(12.0,Length.MICROMETERS)
# Print the reference wavelength in millimetres.
print myImage.getWavelength(Length.MILLIMETERS)
# Print the units being used
print myImage.getUnit()
# Print the intensity at pixel position 30, 35
print myImage.getIntensity(30, 35)
# Display the image
Display(myImage)
```

To add exposure and error maps, use the `setExposure` and `setNoise` methods, each taking a `Numeric2d` map as input. Use `getExposure` and `getNoise` to retrieve these maps.

In a similar vein to the above, you can create a `SimpleCube` to store three-dimensional images (or multiple stacked 2D images). The `SimpleCube` can also include error, flag and exposure maps, which must also be 3D arrays. Only a single WCS can be applied to the `SimpleCube`: for example, it is not possible to provide different WCS's for each image in an image stack.

To create a `SimpleCube` you need to import a `Double/Int3d` object. This is shown by the following example, which reuses the `myImage`, `myQuant` and `myWcs` variables from the previous example. The `d3` "cube" is just a stack of three copies of the same image.

```
l1 = myImage.getImage()
l2 = myImage.getImage()
l3 = myImage.getImage()
d3 = Double3d()
d3.append(l1,0) # Append the image along the 0 axis (stacking)
d3.append(l2,0) # Append the same image
d3.append(l3,0) # Append the same image
# Create the SimpleCube
myCube = SimpleCube(description="Veil nebula in 3D", \
  unit = myQuant, image = d3, wcs = myWcs)
# Print the units
print myCube.getUnit()
# Print intensity at pixel position 30, 35 in layer (depth) 0, the first layer
print myCube.getIntensity(0, 30, 35)
```

A `SimpleCube` accepts the following components:

- Image: this is the most important field. It contains the flux of the cube, and its dimensions define the dimensions of the other fields. When initialised, this field automatically initialises a WCS containing the needed information.

  The type of this field is a subtype of `AbstractOrdered3dData`, usually `Double3d`.

- Error: contains the error values for the corresponding spaxels. This is an optional field. The unit of the error is the same as the one of the image.

- Exposure: contains a 3d array of the same dimension as the image. It gives the exposure of each pixel: one exposure time per sky position and spectral value.

- Flag: contains the Flag array for all the pixels of the cube.

- Unit: gives the unit of the image itself, i.e. the unit of the flux per pixel.

2.11.1. Spectral cubes

A spectral cube is a set of three-dimensional data, with two *spatial* and one *spectral* dimensions.

Conceptually, a spectral cube can be seen in three ways:

- As a stack of *monochromatic images*, like the `SimpleCube` created in the previous example.

- As a cloud of points, when at least one of the axes is not regularly sampled.

- As a set of *spatially related spectra*.

In the HCSS, as we have just seen, a spectral cube can be represented by a `SimpleCube`. However, this class is generally aimed at three-dimensional data, not necessarily with a spectral dimension.

A more specialized class is `SpectralSimpleCube`. This product is an evolution of `SimpleCube`, and as such it includes all its features, such as the error and exposure maps seen before. The main difference is that reading an (*x, y*) position in a `SpectralSimpleCube` will return a `Spectrum1d`, while doing the same with a `SimpleCube` will return a generic one-dimensional array of flux values.

It is possible to convert a `SimpleCube` to a `SpectralSimpleCube`:

```
mySpecSimpleCube = SpectralSimpleCube(mySimpleCube)
```

As for all Herschel products, both can be exported to FITS format.

Note

The *Cube Spectrum Analysis Toolbox* (CSAT) is a handy graphical utility available within HIPE to display and manipulate in detail spectral cubes. It is described in detail in the *Data Analysis Guide*.

Note that spectral cubes must have a valid WCS to be accepted by the CSAT. For more information about adding WCS see Section 2.13.

2.12. Importing spectral cubes from external applications

The following two scripts show how to create spectral cubes from data produced with other applications.

The first script opens a FITS file originally created with NOAO-IRAF from IRAS.

```
homefolder ="/home/agueguen/_WorkHipe/"
myfitsfilename ="katrina_N1569.fits"
from herschel.ia.io.fits import FitsArchive
# force the hipe fits reader to read a non-HIPE FITS file
fits = FitsArchive(reader = FitsArchive.STANDARD_READER)
fits_N1569 =fits.load(homefolder+myfitsfilename)  # Read the file
```

```
print fits_N1569.class   ## this is a Product which contain -:
print -"   --------    -"
print fits_N1569
print -"   --------    -"
PrimImage = fits_N1569["PrimaryImage"]

print -"PrimImage.class=", PrimImage.class          # arrayDataset
print -"PrimImage.data.class=", PrimImage.data.class  # Float3d array

dd= Double3d(PrimImage.data)   # creation of a Double3d from the original Float3d
Simcube=SimpleCube()           # creation of the SimpleCube
Simcube.setImage(dd)           # Setting up the image

mywcs=Simcube.getWcs()

# to read the header of the fits file,
# access it via    <Product>.meta[i].getValue()
#for i in katrina_N1569.meta.keySet():
# print -"meta = -"+i
# print -"Value ="
# print katrina_N1569.meta[i].getValue()

# initialisation of the wcs
for i in fits_N1569.meta.keySet():
 print -"meta = -",i, -" value -",fits_N1569.meta[i].getValue()
 mywcs.setParameter(i,fits_N1569.meta[i].getValue() -,"automatically copied")

#usual values for various keywords, can be updated afterwards with commands like:
# java style
#mywcs.setCunit1("arcsec")
#mywcs.setCunit2("Arcsec")
#mywcs.setCtype1("RA--TAN")
#mywcs.setCtype2("DEC-TAN")
# -......
# jython style
#mywcscunit1="arcsec"
#mywcscunit2="Arcsec"
#mywcsctype1="RA--TAN"
#mywcsctype2="DEC-TAN"
# Both ways are working in the HIPE editor

# If some parameters are missing they should be added manually:
# in this file cunit3 is missing we add it.
mywcs.cunit3 ="angstrom"

# To check for missing keywords a method exists:
mywcs.isCompleteWcs()

# Finally we update the wcs of the simplecube
Simcube.wcs = mywcs
# Simcube can now be opened with the CubeSpectrumAnalysis toolbox.
```

The second script imports a cube from Sinfoni.

```
localfolder = -"/home/agueguen/_WorkHipe/fitsfiles/"
sinfoniefilename = -"cube_sinfonie_UDF3538_bourneau.fits"
# Manual import of FITS files
fits = FitsArchive()

# Sinfoni data converted in a SimpleCube
fits = FitsArchive(reader = FitsArchive.STANDARD_READER)
sinfoniproduct = fits.load(localfolder+sinfoniefilename)
print sinfoniproduct.class   ## This is a Product which contains:
print -"   --------    -"
print sinfoniproduct
print -"   --------    -"
PrimImage = sinfoniproduct["PrimaryImage"]

print -"PrimImage.class=", PrimImage.class          # arrayDataset
print -"PrimImage.data.class=", PrimImage.data.class # Float3d array
```

```
dd= Double3d(PrimImage.data) # Creation of a Double3d from the original Float3d
sinfoniCube=SimpleCube()      # Creation of the SimpleCube
sinfoniCube.setImage(dd)      # Setting up the image

mywcs=sinfoniCube.getWcs()

## initialisation of the wcs
for i in sinfoniproduct.meta.keySet():
 if (sinfoniproduct.meta[i].getType() == java.lang.Long):
    mywcs.setParameter(i,sinfoniproduct.meta[i].getValue()*1. -,"automatically
copied but corrected")
 else:
    mywcs.setParameter(i,sinfoniproduct.meta[i].getValue() -,"automatically copied")
# In the previous for loop we convert long values to double
# to comply with HCSS requirements.
# When you have problems importing data you should have a look at
# the type of data coming from the FITS file and convert it
# it needed.
# To do this you can use
# sinfoniproduct.meta[i].getType()
# sinfoniproduct.meta[i].getClass()

# Usual values for various keywords,
# can be updated after if needed with commands like:
# java style
#mywcs.setCunit1("arcsec")
#mywcs.setCunit2("Arcsec")
#mywcs.setCtype1("RA--TAN")
#mywcs.setCtype2("DEC-TAN")
# -......
# jython style
#mywcsunit1="arcsec"
#mywcsunit2="Arcsec"
#mywcsctype1="RA--TAN"
#mywcsctype2="DEC-TAN"

# To check for missing keywords a method exist:
print mywcs.isCompleteWcs()
# Another way is to print the WCS or open it in the spectrum explorer.
print mywcs.isValid()
# Finally we update the WCS of the Simple Cube
sinfoniCube.wcs=mywcs
# Simcube can now be opened with the Cube Spectrum Analysis Toolbox.
```

2.13. Assigning a World Coordinate System to images and cubes

You can assign WCS information to images and cubes. The World Coordinates System (WCS) describes the coordinates of a `SimpleImage` or `SimpleCube`. It makes it possible to convert image coordinates to world coordinates and the other way around. The WCS can have a lot of parameters, as defined in the FITS standard:

- naxis: the number of axes

- crval1: First coordinate of the centre

- crval2: Second coordinate of the centre

- crpix1: Reference pixel X coordinate

- crpix2: Reference pixel Y coordinate

- cdelt1: Pixel scale of axis 1. Step per pixel or number of degrees per pixel along x-axis when converting to Sky Coordinates. These parameters are no longer used in modern Wcs definition, but are included in the CDi_j matrix.

- cdelt2: Pixel scale axis 2. Step per pixel or number of degrees per pixel along y-axis when converting to Sky Coordinates. These parameters are no longer used in modern Wcs definition, but are included in the CDi_j matrix.

- ctype1, ctype2: Projection type name. This can be "LINEAR", "PIXEL" or the FITS convention. The default value for ctype1 and ctype2 is "LINEAR". When using the FITS convention, the first four characters are:

  - RA-- and DEC- for equatorial coordinates

  - GLON and GLAT for galactic coordinates

  - ELON and ELAT for ecliptic coordinates

  The next four characters describe the projection. Possibilities are:

  - -AZP: Zenithal (Azimuthal) Perspective

  - -SZP: Slant Zenithal Perspective

  - -TAN: Gnomonic = Tangent Plane

  - -SIN: Orthographic/synthesis

  - -STG: Stereographic

  - -ARC: Zenithal/azimuthal equidistant

  - -ZPN: Zenithal/azimuthal PolyNomial

  - -ZEA: Zenithal/azimuthal Equal Area

  - -AIR: Airy

  - -CYP: CYlindrical Perspective

  - -CAR: Cartesian

  - -MER: Mercator

  - -CEA: Cylindrical Equal Area

  - -COP: COnic Perspective

  - -COD: COnic equiDistant

  - -COE: COnic Equal area

  - -COO: COnic Orthomorphic

  - -BON: Bonne

  - -PCO: Polyconic

  - -SFL: Sanson-Flamsteed

  - -PAR: Parabolic

  - -AIT: Hammer-Aitoff equal area all-sky

  - -MOL: Mollweide

  - -CSC: COBE quadrilateralized Spherical Cube

- -QSC: Quadrilateralized Spherical Cube

- -TSC: Tangential Spherical Cube

- -NCP: North celestial pole (special case of SIN)

- -GLS: GLobal Sinusoidal (Similar to SFL)

- Other types are also possible (for example TEMP for temperature.)

  - cunit1: The Unit of Axis 1.

  - cunit2: The Unit of Axis 2.

  - epoch: Epoch of coordinates.

  - Radesys: The reference frame, default value is "ICRS".

  - pc1_1: Element (1,1) of the linear transformation matrix. The pc1 and pc2 parameters are no longer used in modern Wcs definition, but are together with CDELT1 and CDELT2 included in the CDi_j matrix.

  - pc1_2: Element (1,2) of the linear transformation matrix.

  - pc2_1: Element (2,1) of the linear transformation matrix.

  - pc2_2: Element (2,2) of the linear transformation matrix.

  - cd1_1: Element (1,1) of the corrected linear transformation matrix.

  - cd1_2: Element (1,2) of the corrected linear transformation matrix.

  - cd2_1: Element (2,1) of the corrected linear transformation matrix.

  - cd2_2: Element (2,2) of the corrected linear transformation matrix.

With a third dimension the following also applies:

- ctype3: Description of what the 3rd axis represents, for instance Wavelength, Time, M1 Temperature, and so on.

- cunit3: The Unit of Axis 3.

- crval3: [Optional - in case of equidistant 3rd dimension]. Wavelength, time, ... of reference layer; unit : length, time, ...

- crpix3: [Optional - in case of equidistant 3rd dimension] Reference layer index

- cdelt3: [Optional - in case of equidistant 3rd dimension] Scale in 3rd dimension - unit: length, time, ...

- PC elements:

  - pc1_3: Element (1,3) of the linear transformation matrix.

  - pc2_3: Element (2,3) of the linear transformation matrix.

  - pc3_1: Element (3,1) of the linear transformation matrix.

  - pc3_2: Element (3,2) of the linear transformation matrix.

  - pc3_3: Element (3,3) of the linear transformation matrix.

To create a WCS object that can be assigned to an image you can use something like the following.

```
# Create the WCS object, units in degrees by default
myWcs = Wcs(crpix1 = 29, crpix2 = 29, crval1 = 30.0, crval2 = --22.5, \
    cdelt1 = 0.0004, cdelt2 = 0.0004, cunit1 = -"DEGREES", \
    cunit2 = -"DEGREES", ctype1 = -"RA---TAN", ctype2 = -"DEC--TAN")
# Check whether the WCS is valid
print myWcs.valid
# Assign the world coordinates to our image
myImage = SimpleImage(description = -"Veil nebula", wcs = myWcs)
# You can then obtain the world coordinates at any pixel
print myImage.getWcs().getWorldCoordinates(31,31)
# This provides an array of sky coordinates in degrees.
# We can get the intensity at a given WCS position
# First put an image in...
importImage(myImage, -"../data/ia/demo/data/ngc6992.jpg")
# Get the intensity at a given WCS position.
print myImage.getIntensityWorldCoordinates(30.0012, --22.498)
```

For the `SimpleCube` and `SpectralSimpleCube` objects you can do this almost identically. Using the `d3` cube defined in a previous example:

```
# Create WCS object, units in degrees by default
myWcs = Wcs(crpix1 = 29, crpix2 = 29, crval1 = 30.0, crval2 = --22.5, \
    cdelt1 = 0.0004, cdelt2 = 0.0004, cunit1 = -"DEGREES", \
    cunit2 = -"DEGREES", ctype1 = -"RA---TAN", ctype2 = -"DEC--TAN")
# Create the cube
myCube = SimpleCube(description="Veil nebula", image = d3, wcs = myWcs)
# Add third axis (WCS is created with two axes by default)
myWcs.NAxis = 3
# Add quantities related to the third axis
myWcs.crval3 = 300.0
myWcs.crpix3 = 0
myWcs.cdelt3 = 0.001
myWcs.ctype3 = -"Wavelength"
myWcs.cunit3 = -"MICROMETERS"
# You can obtain the world coordinates at any pixel on the image.
print myCube.getWcs().getWorldCoordinates(31,31)
# Get the intensity at a given WCS position. We need three
# arguments now, with the first argument being the layer number (depth)
# from which we want the intensity measure. Count starts from 0.
print myCube.getIntensityWorldCoordinates(0,30.0012, --22.498)
```

If the third axis of the cube is irregularly sampled, you can define an `imageIndex` array with the sampling values of each layer along the axis. Such array would replace the values of the `crval3`, `crpix3` and `cdelt3` parameters:

```
# Create WCS object, units in degrees by default
myWcs = Wcs(crpix1 = 29, crpix2 = 29, crval1 = 30.0, crval2 = --22.5, \
    cdelt1 = 0.0004, cdelt2 = 0.0004, cunit1 = -"DEGREES", \
    cunit2 = -"DEGREES", ctype1 = -"RA---TAN", ctype2 = -"DEC--TAN")
# Create the cube
myCube = SimpleCube(description="Veil nebula", image = d3, wcs = myWcs)
# Add third axis (WCS is created with two axes by default)
myWcs.NAxis = 3
# Add quantities related to the third axis
myWcs.ctype3 = -"Wavelength"
myWcs.cunit3 = -"MICROMETERS"
# Add the imageIndex array
from herschel.share.unit.Length import MICROMETERS
wavelengths = Double1d([20.0, 45.0, 100.0])
myWcs.setImageIndex(wavelengths, MICROMETERS)
```

To check whether the third axis is regularly sampled, the following will return 1 if true, 0 if false:

```
print myWcs.equidistantInZ
```

2.14. Products

Let us briefly run through what we have covered so far. We started with simple arrays in Section 2.3, went on with multidimensional arrays in Section 2.6 and introduced array datasets in Section 2.7. Then it was time for table datasets in Section 2.8 and composite datasets in Section 2.9. As you can see, every object we have examined acted as a container for the previous ones. Now we complete the journey by introducing the highest level of them all, the *Product*.

A Product is an object containing a set of metadata entries (some of which are mandatory) and one or more datasets. The mandatory metadata values are `description`, `creator`, `creationDate`, `instrument`, `startDate`, `endDate`, `modelName` and `type`. They will be automatically added whenever you create a new product:

```
myProduct = Product()      # Creating a new, empty Product
print myProduct.meta       # Printing its metadata
print myProduct.getMeta()  # Same thing, -"Java style"
```

2.14.1. Mandatory Parameters in Products

As you can see some entries are already set to meaningful values, others are set to `Unknown`. You can now modify the mandatory metadata and add as many new entries as you wish. There are so-called "setter" methods for setting values of the mandatory metadata, which currently includes a description, the creator, an instrument, model name of the instrument in use and type, as shown below:

```
myProduct.setDescription("My SPIRE product")
myProduct.setCreator("Myself")
myProduct.setInstrument("SPIRE")
myProduct.setModelName("PFM")
myProduct.setType("UM")
```

Alternately, these can be set using

```
myProduct.creator = -"Myself"
myProduct.instrument = -"SPIRE"
etc...
```

Finally, we can include many of these settings on a single line

```
myProduct=Product(creator="Myself", instrument="SPIRE", \
  description="My SPIRE product", modelName="PFM", type="UM")
```

2.14.2. Setting Date Information

The creation, start and end dates for a Product need to be expressed in terms of a FineTime. If all of these are the current date then we can convert a Java date to a FineTime and include it as metadata in our product. For example:

```
from herschel.share.util.fltdyn.time import FineTime

myProduct.setCreationDate(FineTime(java.util.Date()))
myProduct.setStartDate(FineTime(java.util.Date()))
myProduct.setEndDate(FineTime(java.util.Date()))
```

Because the `startDate`, the `endDate` and the `creationDate` are mandatory metadata parameters, they are set to the current date and time at the moment when the product is created. If those dates are not the current date then it is possible to set it up using UTC or TAI representation of a calendar day (see e.g. Section 6.2), like it is shown in the following example:

```
from herschel.share.fltdyn.time import *
```

```
formatter = SimpleTimeFormat(TimeScale.UTC)
timeUtc = formatter.parse("2008-01-31T12:35:00.0Z")    # Z at the end is mandatory
for UTC

formatter = SimpleTimeFormat(TimeScale.TAI)            # or just SimpleTimeFormat()
timeTai = formatter.parse("2008-01-31T12:35:00.0TAI")  # TAI at the end is
mandatory for TAI

myProduct.setCreationDate(timeUtc) # or
myProduct.setCreationDate(timeTai)
```

Note that the two previous dates, represented as FineTime, are different:

```
print timeUtc    # 2008-01-31T12:35:33.000000 TAI (1580474133000000)
print timeTai    # 2008-01-31T12:35:00.000000 TAI (1580474100000000)
```

2.14.3. Additional Metadata

Now, to add, modify and read additional metadata:

```
myProduct.getMeta().set("Here goes a name", StringParameter("Here goes a value"))
print myProduct.meta["Here goes a name"]
# {description="", string="Here goes a value"}
```

In the example above we set a name and a value for the metadata. In this case the value was represented by a `String` object, but as you already now you can also assign other types of values with `LongParameter`, `DoubleParameter`, `BooleanParameter` and `DateParameter`.

2.14.4. Inserting and Getting Datasets from a Product

But how do you insert and get the contents of the datasets in a product? You can use the `getDefault()` method to get the first dataset stored in the product, or the `get()` method to get any stored dataset, whose name you have to provide as argument. The name is a string assigned when the dataset is first inserted into the product. Here is an example:

```
myTable = TableDataset()
myTable.setDescription("This is a Table Dataset")
myComposite = CompositeDataset()
myComposite.setDescription("This is a Composite Dataset")
myProduct.set("oneDataset", myTable) # We have to give a name to every
                                     # dataset we insert
myProduct["anotherDataset"] = myComposite     # Jython style to add a dataset
myProduct.set("anotherDataset", myComposite) # Java style
print myProduct.getDefault() # As you will see from the description,
                             # this is the Table Dataset
print myProduct["anotherDataset"]     # Getting the Composite Dataset,
                                      # Jython style...
print myProduct.get("anotherDataset") # -...and Java style
```

Instead of just printing out the datasets you get, you can assign them to variables and execute other operations on them. To see how to explore the contents of datasets please refer to the previous sections of this chapter.

If you are not a fan of the command line you can use the handy Dataset Inspector tool to view and manipulate datasets and products. This tool is described in the *Data Analysis Guide*.

2.14.5. Product history

The Product history is generated or updated whenever a task is run on a product. It contains the tasks which have been run to generate the product (including used parameters), as well as the used calibration files and the track and build number of the used build.

You can retrieve the history of a product in Jython as follows:

```
history = product.history
```

A simple `print` shows which tasks, build numbers and calibration files have been used:

```
HIPE> print history
```

Further interesting function of the history are the following:

- Get the history as a Jython script:

```
script = product.history.script
product.history.saveScript("script.py")
```

- Find out if a certain task has been run (useful in a task which depends on another task):

```
if not product.history.isTaskPerformed("someTask"):
      print -"You have to run someTask first!"
      exit
```

Calibration file appear in the history with identifiers constructed from meta keywords. How this is done depends on the instrument:

- For PACS the identifier is constructed from the meta keywords `calFileId`, `modelName` and `calFileVersion` as follows:

```
Common|Photometer|Spectrometer_calFileId_modelName_calFileVersion
```

Also, if you want to introduce your own handcrafted PACS calibration file you should change especially the `calFileId` keyword to make sure that this is visible in the history.

- For SPIRE the `fileName` meta keyword is used to identify calibration files.

- For HIFI no standard has been implemented yet.

Chapter 3. The *Numeric* library

3.1. Introduction

This chapter describes how to use the *Numeric* library in your Jython scripts. For further details of the functions provided, or use of the library from Java programs, please see the API documentation for herschel.ia.numeric in the *Developer's Reference Manual*.

The purpose of the numeric library is to provide an easy-to-use set of numerical array classes (programs) and common numerical functions. The library also supports arrays of booleans and strings.

3.2. Getting started

The numeric library is available to you on starting a HIPE session. Basic setup and arithmetic manipulation of array datasets of various types are discussed in Chapter 2.

3.3. Basic numeric array arithmetic

DP numeric arrays support arithmetic operations that are applied element-by-element. For example:

```
y = Double1d.range(5)                  # [0.0,1.0,2.0,3.0,4.0]
print y * y * 2 + 1                     # [1.0,3.0,9.0,19.0,33.0]
```

This is much simpler (and runs much faster) than writing an explicit loop in Jython. **It is important to appreciate that the '+' operator does not concatenate arrays, as it does with Jython arrays.** For example:

```
# Adding Jython arrays
print [0,1,2,3] + [4,5,6,7]                      # [0, 1, 2, 3, 4, 5, 6, 7]

# Adding DP numeric arrays
print Double1d([0,1,2,3]) + Double1d([4,5,6,7])        # [4.0,6.0,8.0,10.0]

# Concatenate two DP numeric arrays
print Double1d([0,1,2,3]).append(Double1d([4,5,6,7]))
# [0.0,1.0,2.0,3.0,4.0,5.0,6.0,7.0]

# Adding Jython arrays to DP numeric arrays
print [0,1,2,3] + Double1d([4,5,6,7])        # [4.0,6.0,8.0,10.0]
print Double1d([0,1,2,3]) + [4,5,6,7]        # [4.0,6.0,8.0,10.0]
```

All arrays currently support the following arithmetic operators:

```
+, --, *, -/, %, **
```

Note that the 'modulo' operator '%' provides the normal Jython semantics for this operation, which is not the same as that of the Java '%' operator. The Jython definition is more consistent with the mathematical notion of congruence for negative values.

The following relational operators are also provided, which return a Bool1d array:

```
<, >, <=, >=, ==, -!=
```

For example:

```
y = Double1d([0,1,2,3,4])
print y > 2                            # [false,false,false,true,true]
```

3.4. Numeric functions and lambda expressions

In DP, functions can be applied very simply as follows:

```
print SQRT(16) # 4.0 (applied to a scalar)
y = Double1d([1,4,9,16])
print SQRT(y) # [1.0,2.0,3.0,4.0] (applied to a DP numeric array)
```

As shown by this example, functions on scalars (such as SQRT) are implicitly mapped over each element of an array. Functions may be combined with arithmetic operators to perform complex operations on each element of an array:

```
t = Double1d([1,2,3,4])
print SIN(1000 * t * (1 + -.0003 * COS(3 * t)))
# [0.6260976237441638,0.5797470124743422,0.8629107307631398,
#-0.9811675382238753]
```

The **names of functions in the numeric library have ALL LETTERS** capitalised. This is to avoid ambiguity, as Jython already defines certain functions, such as 'abs', which are not applicable to our DP numeric arrays.

There are various types of functions in the numeric library:

```
y = Double1d([1,2,3,4])

print SQRT(4) # double->double
print SQRT(y) # double->double (mapped)
print REVERSE(y) # Double1d->Double1d
print MEAN(y) # Double1d->double
```

It is possible to define **new functions** as *lambda expressions* in Jython and apply them to DP numeric arrays. For example:

```
y = Double1d([1,2,3,4])

f = lambda x: x*x + 1 #take the given array, call it -'x' and
#return the value x^2 +1 to an array called f.

print f(y) #[2.0,5.0,10.0,17.0]. Each element of y was
#taken --> x then each element was squared
#plus 1 added.
```

However, in this case, it's much easier and faster to do this with array operations.

```
print y * y + 1
```

Lambda expressions are not as fast as the standard Java functions provided by the numeric library, but this is often not a problem. Where performance is an issue, new functions can be defined in Java (see the JavaDoc of the herschel.ia.numeric library).

More complex functions (equivalent to subroutines) can be created using the **def** command, which is discussed in Section 1.11.

3.5. Selection, data filtering and masking methods

The numeric library provides operations, such as 'filter', which allows the selection of array elements based on a given criterion (e.g., element with values between 3 and 6). There is no 'map' operation because mapping is implicit with the array style of processing.

The 'filter' method returns a Double1d array. The selection criterion for the filter method MUST be declared using a lambda function:

```
u = Double1d.range(10)
print u.filter(lambda x: x>3 and x<6)
```

Note: The Jython filter operation can be used but returns a Jython array:

```
print filter(lambda x: x>3 and x<6, u)
# __class__ returns org.python.core.PyList
print filter(lambda x: x%2==1, u)
```

Jython *list comprehensions* can be used but also return Jython arrays:

```
print [x for x in u if x>3]
print [x*x for x in u if x>3 and x<6]
print Double1d([x*x for x in u if x>3 and x<6])
#this last now provides us with a numerical array as we have also
#translated into a Double1d array.
```

The SQUARE function could equally have been applied:

```
print u.filter(lambda x: x>3)
print SQUARE(u.filter(lambda x: x>3 and x<6))
```

Warning

If a lambda expression is applied to an array, remember that it is applied to the entire array and not mapped over the elements. This can lead to unexpected behaviour as in the following example:

```
u = Double1d.range(10)
print (lambda x: x>2 and x<4)(u)
# [true,true,true,true,false,false,false,false,false,false]
```

This is equivalent to the following:

```
u > 2 and u < 4
```

The expression 'u>2' results in a Bool1d array. The Jython 'and' treats this as 'true', as it is a non-empty list, and returns the result of the second expression 'u<4', which is not the intended result.

One way of overcoming this problem is to use the '&' operator instead of 'and' to give the intended result:

```
print (lambda x: (x>2) & (x<4))(u)
# [false,false,false,true,false,false,false,false,false,false]
```

Warning

This shows how the '&' operator and the 'and' operator are not identical operators.

If you wish to select elements of an array based on a given criterion then we can find out 'where' in a sequence of data a certain type resides (e.g., at what position the maximum value of an array occurs) and how to get the data that fits your selection.

For example, the where method returns the array indices of elements that satisfy a predicate often given as a lambda function. The input to the where method is a Boolean array. This differs from the filter where the actual elements themselves are obtained. Using the modulo function (%) we can find where within an array odd values occur.

```
y = Double1d([2,6,3,8,1,9])
```

```
print y.where(y%2==1) # [2,4,5] indices of odd elements
```

Now return the actual elements, which can be done in three ways

```
print y[y.where(y%2==1)] # [3.0,1.0,9.0]
print y.filter(lambda y: y%2==1) # [3.0,1.0,9.0]
print y.get(y%2==1) # [3.0,1.0,9.0]
```

Predicates support standard jython operators such as `not`, `and` and `or`:

```
y = Double1d([1,2,3,4])
print y.where(lambda x: x<3 and x>1) # [1]
```

Java/C-style logical operators '!', '&&', and '||' are not allowed.

It can be useful to have the indices, rather than the values, when there are two or more arrays with a predicate applied to one of them. For example:

```
x = Double1d([5,6,7,8])
s = y.where(y%2==1)
print x[s] + y[s] # [6.0,10.0]
```

The **'where' function** can also be used to set values:

```
s = y.where(y%2==1)
y[s] = 0 # Set all matching elements to 0
print y # [0.0,2.0,0.0,4.0]
y[s] = [9,8] # Set matching elements using an array of values
print y # [9.0,2.0,8.0,4.0]
```

> **Note**
>
> You can't use the `where` function like this:
>
> ```
> a=Double1d.range(10)
> b=a.where(a < 3)
> print b[0]
> print b[0:2]
> print a[b[0]]
> ```
>
> The last three lines will give an error. Technically, this is because b is a `Selection` object rather than a Jython or Numeric array. For the above to work you need to convert it to `Int1d`:
>
> ```
> c = b.toInt1d()
> print c[0] # Now these three lines will work
> print c[0:2]
> print a[c[0]]
> ```

The **'get' method** enables you to grab individual elements or a subset of element values from an array. It requires the input of a Boolean array (e.g., a mask). Along with getting individual elements, there are three other forms. One enables you to select element values based on a `Bool1d` mask:

```
y = Double1d([5,7,8,9])
mask = Bool1d([0,0,1,0])
x = y.get(mask) # x == [8.0]
```

The second form enables you to select on a set of indices, contained in a `Selection` object:

```
indices = Selection(Int1d([2,3]))
x = y.get(indices) # x == [8.0,9.0]
```

The third form enables you to select elements from a range, specified by a `Range` object:

```
range = Range(2,4)
x = y.get(range) # x == [8.0,9.0]
```

It is possible to combine 'get' calls to perform the same operation as a compound IDL WHERE execution. Let's set up a few arrays first:

```
a = Double1d([1, 2, 3, 4, 5, 6])
b = Double1d([2, 3, 4, 5, 6, 7])
c = Double1d([3, 4, 5, 6, 7, 8])
```

The following operations on the three arrays are the equivalent of the IDL WHERE statement 'where(a ge 2 and b lt 6 and c gt 5)':

```
q = (a >= 2) & (b < 6) & (c > 5)
x = a.get(q),b.get(q),c.get(q) # x == ([4.0], [5.0], [6.0])
```

3.6. Array access and slicing

The numeric package introduces the following square brackets notation:

```
[i_0,...,i_n-1]
```

where each element is separated by a comma, and the number of elements must be equal to the rank of the array. Arrays are zero-based which means the first element of an array has index 0 (zero) and the index of the last element of an array is array.length()-1.

In addition the package supports the colon (:) notation to designate a slice. A slice is a range of indices defined as i:j, where i is the starting index and inclusive, and it is zero if not specified. The ending index j is exclusive and it is equal to array.length() if not specified and array.length()-j if negative.

The following example illustrates the access to elements in a multi-dimensional array and the use of slices. More examples can be found in the section on Multi-Dimensional Arrays.

```
# define something that is like a rectangular 2x3 array:
#   1 2 3
#   4 5 6
x=Int2d([[1,2,3],[4,5,6]])# Int1d can swallow the jython sequence.
print x                    # [[1,2,3],[4,5,6]]
print x[1]                 # 2 (second element of the first row)
print x[1,:]               # access a row i.e. [4,5,6]
print x[1,1]               # access an individual element i.e. 5
print x[:,:]               # [[1,2,3],[4,5,6]]
print x[:,1]               # access a column i.e. [2,5]
```

3.7. Making sense of logical operators

Here we try to guide you through the jungle of logical operators you are likely to encounter when using DP.

First of all, since Jython is embedded in DP, it won't surprise anyone that the **Jython logical operators** and, or and not are available. These work like normal Boolean operators (see Appendix C for more details), but using them with arrays (both the native Jython ones and those from the DP Numeric package) can give unexpected and seemingly inexplicable results. See below for an example. The important thing to keep in mind is that these operators do *not* work on an element-by-element basis when applied to arrays, but they evaluate the entire array at once.

Another tool coming straight from the Jython language are the **bitwise operators**, represented by the symbols &, | and ^. See again Appendix C for more details. The possible source of confusion here

is that these symbols can be used with Numeric arrays (e.g. `Int1d`, `Bool3d` etc.), but what you get is *not* a bitwise comparison. Instead, these operators perform the usual boolean comparisons, but this time working element by element. Precisely what `and`, `or` and `not` do *not* do.

Finally, Numeric array classes have the `and`, `or` and `xor` *methods* acting like boolean *operators* working element by element. An example will hopefully clarify the differences among all the operators described here:

```
jythonOne = [1, 0, 0, 1]
jythonTwo = [0, 0, 1, 1]
numericOne = Bool1d(jythonOne)
numericTwo = Bool1d(jythonTwo)
print jythonOne and jythonTwo
## [0, 0, 1, 1] # jythonOne is not empty so it is treated as true, which means that
                # jythonTwo is evaluated and returned
print numericOne and numericTwo
## [false,false,true,true] # Same thing as with the Jython native arrays
print jythonOne & jythonTwo
## Here an error is returned
print numericOne & numericTwo
## [false,false,false,true] # Here the operator works element by element
print numericOne.and(numericTwo)
## [false,false,false,true] # Same thing as the & operator
```

3.8. Advanced tips for improved performance

The underlying array operations and functions are very fast, as they are implemented in Java. The overhead of invoking them from Jython is relatively small for large arrays. However, the advanced user may find the following tips useful to improve performance in cases where it becomes a problem.

The arithmetic operations, such as '+', have versions that allow in-place modification of an array without copying. For example:

```
y = Double1d.range(10000)
y = y + 1 # The array is copied
y += 1 # The array is modified in place
```

Copying an array is slow as it involves allocating memory (and subsequently garbage collecting it). For simple operations, such as addition, the copying can take longer than the actual addition.

Function application also involves copying the array. This can be avoided by using the Java API instead of the simple prefix function notation. For example:

```
x = Double1d.range(10000)
x = SIN(x) * COS(x) # This operation involves three copies
x = x.apply(SIN).multiply(x.apply(COS)) # Only one copy
```

When writing array expressions, it is better to group scalar operations together to avoid unnecessary array operations. For example:

```
y = Double1d([1,2,3,4])
print y * 2 * 3 # 2 array multiplications
print y * (2 * 3) # 1 array multiplication
print 2 * 3 * y # 1 array multiplication
```

It is better to avoid explicit loops in the HCSS DP system over the elements of an array. It is often possible to achieve the same effect using existing array operations and functions. For example:

```
sum = 0.0
for i in y:
  sum = sum + i * i # Explicit iteration
```

```
sum = SUM(y * y) # Array operations
```

3.9. Type conversions

Since the numeric library supports different types it would be very convenient to be able to convert an array from one type to another. The numeric library supports both implicit conversion from within jython for all supported dimensions and explicit conversion from one data type to another.

3.9.1. Explicit conversion

Explicit conversion is supported for all data types by constructing a numeric array from another DP numeric array of the same or a different type. Note however that some explicit conversions may result in rounding and/or truncation of the values e.g. an explicit conversion from Long1d to Double1d will reduce the number of significant digits.

```
i = Int1d([1,2,3])                    # [1,2,3]
r = Double1d(i)                       # [1.0,2.0,3.0]
c = Complex1d(r)                      # [(1.0+0.0j),(2.0+0.0j),(3.0+0.0j)]
b = Byte1d(r)                         # [1,2,3]
```

3.9.2. Implicit conversion

Implicit conversions are conversions that can be done by the DP package automatically, provided that such a conversion is a widening operation e.g. from Int1d to Double1d. Implicit narrowing conversions are not allowed and result in an error message as shown below:

TypeError: Conversion of class org.python.core.PyFloat to class java.lang.Long implies narrowing.

The library supports implicit conversions in the following cases:

- access: [...]

- operators: +, -, *, /, ^ and %

- in-line operators: +, -, *, /, ^ and %

The few examples below show allowed implicit conversions.

```
d = Double1d(5)                       # [0.0,0.0,0.0,0.0,0.0]
d[1] = 3                              # [0.0,3.0,0.0,0.0,0.0]
d[1:4] = [-5, 0, 5]                   # [0.0,-5.0,0.0,5.0,0.0]
```

Please note that the DP package considers the conversion from int to float and from long to float/ double as an automatic widening operation, but some of the least significant digits of the value may be lost during the conversion. You will not be notified of this loss of significant digits.

Another thing to notice is that floating point operations will never throw an exception or error. As shown in the following example, a division by zero results in NaN or Infinity.

```
d = Double1d.range(5)
l = Long1d.range(5)
print d/l                            # [NaN,1.0,1.0,1.0,1.0]
print d/SHIFT(l, 1)                  # [0.0,Infinity,2.0,1.5,1.3333333333333333]
```

3.10. Function library

The numeric package includes a library of basic numeric processing functions, which will continue to grow as development of the library progresses.

The functions that are currently available are outlined below. For further details, reference should be made to the **Javadoc documentation** and **demo programs** .

3.10.1. Basic functions

Basic functions applicable to scalars or arrays, and returning scalars or arrays of the same size:

- ABS: Section 2.2

- ARCCOS: Section 2.20

- ARCSIN: Section 2.21

- ARCTAN: Section 2.22

- CEIL: Section 2.50

- COS: Section 2.77

- EXP: Section 2.114

- FIX (not applicable to scalars): Section 2.137

- FLOOR: Section 2.147

- LOG: Section 2.239

- LOG10: Section 2.238

- ROUND: Section 2.339

- SIGNUM: Section 2.355

- SIN: Section 2.368

- SQRT: Section 2.396

- SQUARE: Section 2.397

- TAN: Section 2.405

These are applied in the form

```
b = SIN(a)
```

b will be an array of the same dimension as a or a single value if a is single valued.

Array functions on Double<n>d returning a double:

```
MIN, MAX, MEAN, MEDIAN, RMS, SUM

b = MIN(a) #'b' has the minimum value of the array -'a'.
```

Functions applicable to one-dimensional arrays and returning an array of the same size:

- REVERSE: Section 2.334

Functions applicable to arrays and returning an array of increased rank (number of dimensions):

- REPEAT:

Warning

Many of these functions have lower case equivalents built-in in Jython. Be aware of which one you are using, because their behaviour could differ in some cases, as shown by the example below which creates a table with Not-a-Number's (NANs) in it.

```
tt=Double1d.range(10)
tt[0]=Double.NaN
print max(tt)
# NaN
print min(tt)
# NaN
tt[1]=Double.NaN
tt[0]=1.0
print max(tt) # By using the built-in Jython functions
# 9.0
print min(tt)
# 1.0
print MAX(tt) # By using the DP Numeric functions
# NaN
print MIN(tt)
# NaN
```

3.10.2. Integral transforms

A Discrete Fourier Transform is provided for `Complex1d` arrays. This uses a radix-2 FFT algorithm for array lengths that are powers of 2 and a Chirp-Z transform for other lengths. Future releases might support multi-dimensional arrays, if required, and optimised transforms of real data.

Window functions are provided for reducing 'leakage' effects using the **Hamming** or *Hanning* window.

Example 3.1 shows the generation of a frequency modulated signal, followed by a FFT both with and without windowing:

```
ts = 1E-6          # Sampling period (sec)
fc = 200000        # Carrier frequency (Hz)
fm = 2000          # Modulation frequency (Hz)
beta = 0.0003      # Modulation index (Hz)
n = 5000           # Number of samples

pi = java.lang.Math.PI    # define pi

t = Double1d.range(n) * ts
# t is a 5000 element array holding time values

signal = SIN(2 * pi * fc * t * (1 + beta * COS(2 * pi * fm * t)))
#create the modulated signal with modulation frequency fm and carrier
#frequency fc, t is the array we created above for the time elements.

spectrum = ABS(FFT(Complex1d(signal)))
#spectrum holds the absolute value (ABS) of the FFT of the signal.
#We need to handle these arrays as Complex1d rather than Double1d.

freq = Double1d.range(n) -/ (n * ts)
#The frequency values for the spectrum.

# Repeat with apodizing
spectrum2 = ABS(FFT(Complex1d(HAMMING(signal))))
```

Example 3.1. FFT of a modulated signal , with and without HAMMING smoothing

The *Inverse Fourier Transform* of a Complex1d array (only) "x" can be obtained using, e.g., **inverse = IFFT(x)**.

3.10.3. Power spectrum

With the `PowerSpectrum` class you can create the power spectrum of each column of a Table Dataset. Table dataset that are suitable for power spectrum conversion typically contain a column bearing units of time, plus other columns of quantities from which to compute power spectra. Since real signals sometimes contain unwanted strong excursions, called glitches or spikes, that will dominate the power spectrum, the Task includes a simple de-glitcher, that detects and removes such events from the data stream, replacing them with an average of the surrounding data.

The *Power Spectrum Viewer*, a graphical interface wrapping the functionality of this class, is described in the *Data Analysis Guide*.

You can obtain your power spectra by invoking the `getPowerSpectrum` method on the `PowerSpectrum` class. The method takes the following arguments:

- *table*: the input Table Dataset.

- *flimit*: the inverse cut-off frequency (default 0.1).

- *sigma*: the deglitcher threshold (default 4).

- *deglitch*: boolean, activates the deglitcher if `true` (default).

- *timeColumn*: a `Column` containing time information.

The inverse cut-off frequency determines the length of the intervals into which the data timeline is subdivided before performing the FFT. Each of these datasets is Fourier transformed individually, and the resulting power spectra are quadratically co-added to yield a power spectrum with a better S/N ratio, that is, a higher cut-off frequency will yield a better S/N for the resulting power spectrum.

The sigma value controls a simple sigma/kappa deglitcher, that eliminates all datapoints that are more than sigma (default = 4) times the standard deviation away from the mean. After eliminating these data points the procedure is repeated iteratively until no more data can be discarded.

The `getPowerSpectrum` method has the following variants:

- ```
  getPowerSpectrum(table);
  ```

- ```
  getPowerSpectrum(table,
                   timeColumn);
  ```

- ```
  getPowerSpectrum(flimit,
                   table);
  ```

- ```
  getPowerSpectrum(flimit,
                   table,
                   timeColumn);
  ```

- ```
  getPowerSpectrum(flimit,
                   sigma,
                   table);
  ```

- ```
  getPowerSpectrum(flimit,
                   sigma,
                   table,
                   timeColumn);
  ```

- ```
  getPowerSpectrum(flimit,
                   sigma,
  ```

```
                    deglitch,
                    table);


•   getPowerSpectrum(flimit,
                    sigma,
                    deglitch,
                    table,
                    timeColumn);
```

3.10.4. Convolution

Convolution is currently supported for `Double1d` arrays. A direct convolution algorithm is used, although a future release might implement Fourier convolution to improve the speed for large arrays and large kernels. An example of its use is given in Example 3.2.

```
from herschel.ia.numeric.toolbox.filter.Convolution import *
x = Double1d.range(100)
# Create array [0.0, 1.0, 2.0 -... 99.0]
kernel = Double1d([1,1,1])
#provide kernel for the convolution
f = Convolution(kernel)
#create the convolution
y = f(x)
#apply it to the array x. The result is in array y
```

Example 3.2. Example of the use of the convolution algorithm

This illustrates a general approach with the numeric library i.e. general function objects may be instantiated using parameters to create a customised function which can then be applied to one or more sets of data.

The constructor of the `Convolution` class allows optional *keyword* arguments to be specified, to further customise the function:

- The 'center' parameter allow selection of a causal asymmetric filter for time domain filtering or a symmetric filter for spatial domain filtering.

- The 'edge' parameter controls the handling of edge effects, as well as allowing a choice between periodic (circular) and aperiodic convolution.

The following examples show construction of filters using these options:

Note

Make sure you have input the following import line before trying these out.

```
from herschel.ia.numeric.toolbox.filter.Convolution import *
```

Use zeroes for data beyond edges, causal

```
f = Convolution(kernel, center=0, edge=ZEROES)
```

Circular convolution, causal

```
f = Convolution(kernel, center=0, edge=CIRCULAR)
```

Repeat edge values, causal

```
f = Convolution(kernel, center=0, edge=REPEAT)
```

Use zeroes for data beyond edges with centred kernel

```
f = Convolution(kernel, center=1, edge=ZEROES)
```

Circular convolution with centred kernel

```
f = Convolution(kernel, center=1, edge=CIRCULAR)
```

Repeat edge values with centred kernel

```
f = Convolution(kernel, center=1, edge=REPEAT)
```

3.10.5. Boxcar and gaussian filters

Finite Impulse Response (FIR) filters and symmetric spatial domain filters can be defined by instantiating the `Convolution` class with appropriate parameters. *In addition, special filter functions are provided for Gaussian filters and box-car filters* :

```
from herschel.ia.numeric.toolbox.filter.Convolution import *

f = GaussianFilter(5, center=1, edge=ZEROES)
f = BoxCarFilter(5, center=0, edge=ZEROES)
```

These filters are subclasses of `Convolution` and hence inherit the use of similar keyword arguments.

3.10.6. Interpolation

Interpolation functions are provided for a variety of common interpolation algorithms.

Example 3.3 illustrates the use of the currently available interpolation functions.

```
from herschel.ia.numeric.toolbox.interp import *
# Create the array x [0.0, 1.0, 2.0, -..., 9.0]
x = Double1d.range(10)
print x # [0.0,1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0]
# Create an array y which contains the sine of each element in x
y = SIN(x)
# u contains the values at which to interpolate
u = Double1d.range(80) -/ 10 + 1
print u #[1.0,1.1,1.2,1.3....8.6,8.7,8.8,8.9]
# Linear interpolation
# This sets up the interpolation, linear x-y fit
# Interpolate at specified values
interp = LinearInterpolator(x,y)
# Prints out the values interpolated at each position noted in array u
print interp(u) #[0.8414709848,0.848253629....0.5275664375,0.4698424613]

# NearestNeighbour and CubicSpline interpolation may be performed
# in the same way:

# Cubic-spline interpolation
interp = CubicSplineInterpolator(x,y)

# Nearest-neighbour interpolation
interp = NearestNeighborInterpolator(x,y)
```

Example 3.3. Interpolation functions in DP

The result of the interpolations used in the above example is illustrated in Figure 3.1.
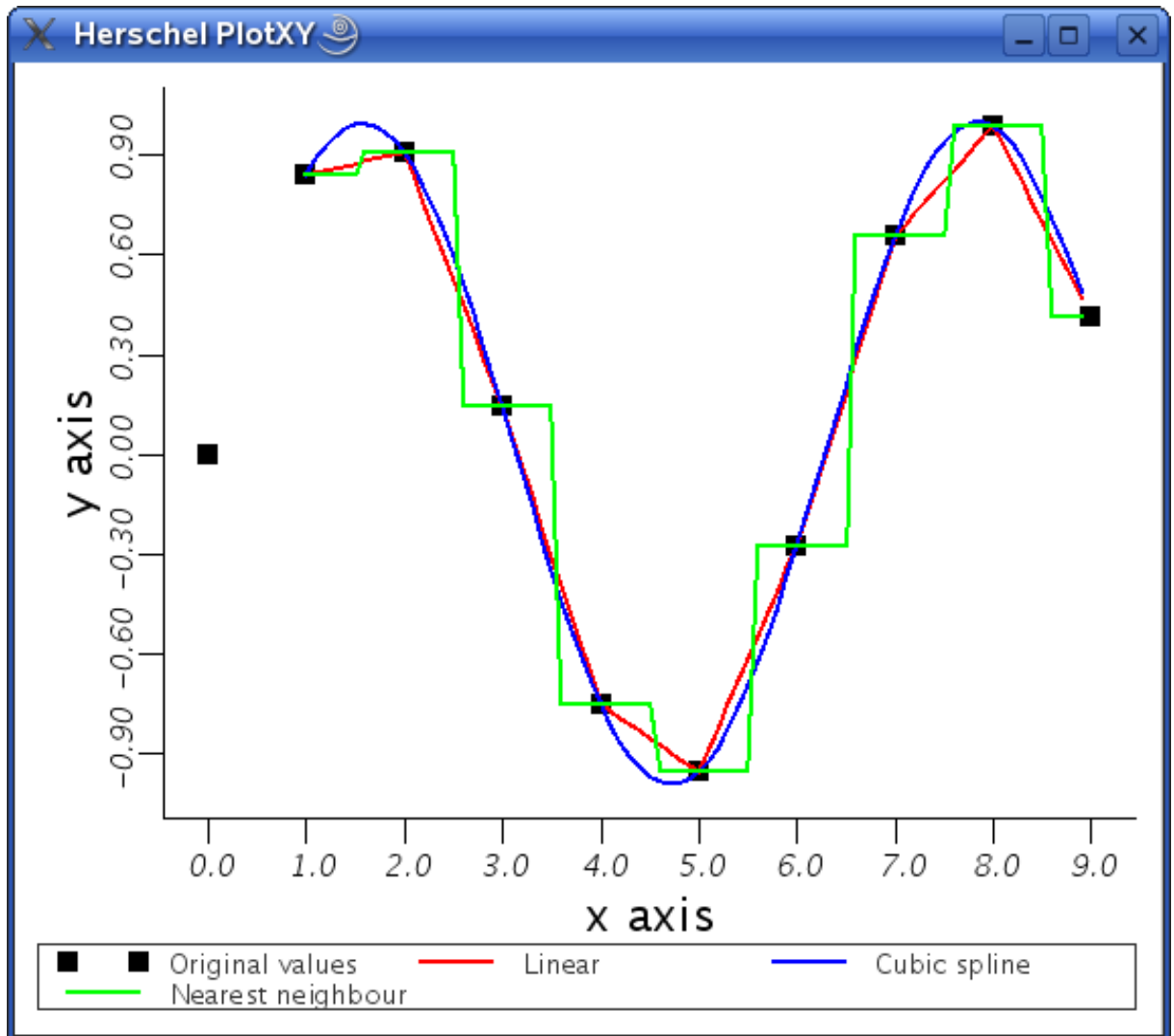
Figure 3.1. Illustration of various forms of interpolation functions.

3.10.7. Data fitting

Here we provide information on the basic linear and non-linear fitting routines available within DP.

3.10.7.1. General approach

Input Data: The fitter package expects your data to be in two datasets that are related to each other. Typically, these are Double1d arrays, e.g.,

```
# Data points: each element in x and y define a data point
x = Double1d.range(12)   # Make x vector (the data positions/channels)
y = Double1d([1.0,1.2,0.9,2.2,3.3,\
 4.5,3.6,2.7,1.8,1.2,1.0,1.1])  # Make y vector (the data values)
```

Model Selection: Fitting means adjusting the parameters of a known function, called *model*, so that it best matches the input data. This toolbox provides some pre-defined linear models as well as non-linear models. Viewing your data will hopefully give you some hints about what function model would reflect your input data. For example, if it seems to be polynomial of a certain degree, you would choose a PolynomialModel.

> **Note**
>
> For the case of non-linear fitters (e.g., used with Gaussians) it is also necessary to provide initial guesses in the form of a parameter set to the model before invoking a fitter. The closer the initial guess for the parameter set to the true values the higher the likelihood that the minimisation will not find a local minimum with wrong/unrealistic parameter estimation.

An example of the use of a linear fitter:

```
# Choose a model: 4th degree polynomial
myModel = PolynomialModel(4)
# Create a fitter and feed it your positions/channels along the array
# (x, a Double1d array) and your model
myFitter = Fitter(x, myModel)
```

Or for a non-linear fitter applied to our array 'x':

```
myModel = GaussModel()
peak = 4.5
channel = 5.5
width=1.0
initialvalues = Double1d([peak, channel, width])
# Apply the initial estimates: peak height, channel position and
# width of gaussian
myModel.setParameters(initialvalues)
# Choose non-linear fitter to use
myFitter = AmoebaFitter(x, myModel) # see later section on available fitters
```

Fit Execution (with and without weights)

```
# Now actually fit the data values at each x position (the y array) to the model
fitresults = myFitter.fit(y)
# Or with associated weights array
fitresults = myFitter.fit(y, yWeights)
```

Results Now the fitter has done its job. We can print the results (`fitresults`) to see the parameters fitted.

```
 print fitresults  # from using the polynomial fitter
# [1.0993589743591299,-1.1096331908843398,0.8923489704745665,
# --0.14688390313399513,0.006825466200470528]
# provides coefficients of the polynomial fit
print fitresults  # from using the Gaussian fitter
# [3.751009700481534,5.353351564022887,2.5098951536394383]
#peak of fit, channel of Gaussian peak, width of Gaussian
```

The fit parameters model are computed and we can start using that model to e.g. re-sample your model fit data:

```
# Re-sample with equally spaced x data points and a finer grid:
xs = Double1d.range(1200) -/ 100     # Re-sampled x positions
ys = myModel(xs)                     # Computed y data points
#a plot of xs versus ys plots out 1200 points with the fit.
```

Statistical Information The above procedure demonstrates how to use the fit package to fit your data against a certain model. However, it does not tell you how good the fit actually is. The fitters provide ways to extract such information from the fit.

```
# After fitting
print myFitter.getChiSquared()        # Goodness of the fit
# e.g., 2.5765684980727577 for Gaussian fit
print myFitter.autoScale()            # How well does the data fit the model.
```

```
# e.g., 0.535056435037312 for Gaussian fit
print myFitter.getStandardDeviation() # Standard deviations for the parameters.
# e.g., [0.30907540430060004,0.24531121048289006,0.2525757390634412]
# for Gaussian fit parameters
print myFitter.getHessian()          # Retrieve the Hessian matrix
es = myFitter.monteCarloError(xs)    # Errors on the resampled datapoints
# es is now an error array with a length the same as -"xs" --- 1200 samples
```

3.10.7.2. Available linear models

There are several models that can be used for linear fitting.

In the descriptions below, the models provide parameter fit values p_0, p_1 ... p_k.

> **Note**
>
> In the following examples the parameter subscripts match the position of the parameter in the output array (`fitsresult` in the previous section). So p_0 will be the first element of the `fitsresult` array, p_1 the second one, and so on.

BinomialModel, which allows for the fitting of a binomial model with two variables -- $f(x,y:p) = \Sigma\ p_k\,x^k\,y^{(d-k)}$, where d is the degree. *Usage: BinomialModel(4) -- provides a binomial model of degree 4.*

PolynomialModel, which allows for the least squares fitting of a polynomial to the data -- $f(x:p) = \Sigma\ p_k\,x^k$. *Usage: PolynomialModel(3) -- provides a third order polynomial fitting of the data.*

SineAmpModel, which allows for the fitting of cosine and sine waves of a given frequency to get amplitudes -- $f(x:p) = p_0\,cos(2\,\pi\,f\,x) + p_1\,sin(2\,\pi\,f\,x)$, where x is the data. *Usage: SineAmpModel(f) -- which provides cosine/sine fits with a frequency, f.*

PowerModel, which allows for the fitting of a power law of order k -- $f(x:p) = p_0\,x^k$. *Usage: PowerModel(3) -- provides a third-order power-law fit*

CubicSplinesModel, which allows for the fitting of a cubic splines with arbitrary knots settings. *Usage: CubicSplinesModel(5) -- provides a cubic splines fit with 5 knots.*

3.10.7.3. Available non-linear models

There are a number of models that can be used for non-linear fitting. For fitting of these models we need initial values (guesses) for parameters labelled p_0, p_1 and p_2 (see example given in the "General Approach" section).

ArctanModel, which allows for the fitting of a general arctan function -- $f(x:p) = p_0\,arctan(p_1\,(x - p_2))$. *Usage: ArctanModel()*

ExpModel, which allows for the fitting of a general exponential function -- $f(x:p) = p_0\,exp(p_1\,x)$. *Usage: ExpModel()*

LorentzModel, which allows for the fitting of a Lorentz function -- $f(x:p) = p_0\,(p_2/((x - p_1)^2 + p_2^2))$. *Usage: LorentzModel()*

PowerLawModel, which allows for the fitting of a general power-law function -- $f(x:p) = p_0\,(x - p_1)^{p_2}$. *Usage: PowerLawModel()*

SincModel, which allows for the fitting of a sinc function -- $f(x:p) = p_0\,sin\,((x - p_1)/p_2)/(x - p_1)/p_2$. *Usage: SincModel()*

SineModel, which allows for the fitting of a general cosine/sine wave -- $f(x:p) = p_1\,cos(2\,\pi\,p_0\,x) + p_2\,sin(2\,\pi\,p_0\,x)$. *Usage: SineModel()*

GaussModel, which allows for the fitting of a 1-D gaussian -- $f(x:p) = p_0\, exp(-0.5\,(\,(x - p_1)\,/\,p_2)^2$, where p_0 is the amplitude, p_1 the x-shift (from zero) and p_2 the sigma of the fit, with initial values of 1.0, 0.0 and 1.0 respectively. Note that *Gauss2DModel* produces a fit to 2D data. *Usage: GaussModel()*

User supplied non-linear function, which allows for fitting a function (linear or non-linear) constructed by the user. This function must be put in a jython class and optionally the user could provide an analytical calculation of the partial derivatives with respect to the parameters (otherwise they are calculated numerically). This is shown in the following example for the following function of four parameters: $f(x:p) = p0/(1+(x/p1)^2)^{p2} + p3$ (the so called beta-profile):

```
from herschel.ia.numeric.toolbox.fit import NonLinearPyModel

class BetaModel(NonLinearPyModel):
# the full 4-parameter beta-model with partial derivatives
# f(x:p) = p0/(1+(x/p1)**2)**p2 + p3
#
  npar = 4
  def __init__(self):
    # Constructor
    NonLinearPyModel.__init__(self, self.npar)
    self.setParameters(Double1d([1,1,-1,1]))
  #
  def pyResult(self,x,p):
    model = p[0]/(1.0 + (x/p[1])**2)**p[2] + p[3]
    return model
  #
  def pyPartial(self,x,p):
    # the partial derivatives
    arg1 = 1.0 + (x/p[1])**2
    dp = Double1d(self.npar)
    #
    dp[0] = 1.0/arg1**p[2] # df/dp0
    dp[1] = 2.0*p[0]*p[2]*x*x/((p[1]**3)*arg1**(p[2]+1.0)) # df/dp1
    dp[2] = --p[0]*Math.log(arg1)/arg1**p[2] # df/dp2
    dp[3] = 1.0 # df/dp3
    return dp
  def myName( self -):
    # Return an explicatory name (String). Optional.
    return -"beta-profile: f(x:p) = p[0]*{1 + (x/p[1])2}^p[2] + p[3]"
```

Once we define the function as shown in the example then we can proceed as before and create a model and then perform the fitting using either the Lavenberg-Marquardt or Amoeba fitters:

```
bm = BetaModel()
bm.setParameters(Double1d([10.0,1.0,-2.0,5.0]))
myfit = LevenbergMarquardtFitter(x, bm) # see section on available fitters below
# or myfit = AmoebaFitter(x, bm)
result = myfit.fit(y)
print result
```

3.10.7.4. Compound and mixed models

It is possible to add two models, e.g. if one wants to fit a spectral line (a Gaussian) on a background (a Polynomial). The resulting model is non-linear.

```
myModel = GaussModel()              # Define a Gaussian
myModel += PolynomialModel(1)       # Add a Polynomial to it of order 1. Only with +=
print myModel.toString()            # Information about the model
```

More models can be added if wished.

3.10.7.5. Available fitters

Fitter. Fitter for linear models. You create a fitter by providing the model assumption and the x points of the data. With that information you compute the parameters within the model by fitting the y data

points. Once the computation of those parameters is done, you can extract statistical information from the fitter. **Syntax: myFitter=Fitter(xDataPoints, model)**

LevenbergMarquardtFitter. Fitter for non-linear models. The LMFitter is a gradient fitter, which means that it goes downhill from the starting location until it cannot go down anymore. There is no guarantee that the minimum found is an absolute or global minimum. If the chisq-landscape is multimodal it ends in the first minimum it finds. See also Numerical Recipes, Ch 15.5. **Syntax: myFitter = LevenbergMarquardtFitter(xDataPoints, model)**

AmoebaFitter. Fitter for non-linear models. The AmoebaFitter implements the Nelder-Mead simplex method. It comes in 2 varieties, one where the simplex simply goes downhill (temperature = 0) and one which implements an annealing scheme. Depending on the temperature, the simplex sometimes takes an uphill step, while a downhill steps always is taken. This way it is able to escape from local minima and it has a better chance of finding the global minimum. No guarantee, however. *AmoebaFitter* is also able to handle limits on the parameter range. Parameters stay within the limits when they are set. See also Numerical Recipes, Ch. 10.4 and 10.9. **Syntax: myFitter = LevenbergMarquardtFitter(xDataPoints, model)**

SingularValueDecompositionFitter. Linear fitter based on Singular Value Decomposition (described in Section 3.10.10. Much more robust in case of (nearly) degenerated models, at the cost of more CPU use. See Numerical Recipes for more information. See also the *User Reference Manual*: Section 2.366.

3.10.7.6. Obtaining a model fit to 1D and 2D data

1D fit example

Example 3.4 shows how a polynomial can be fitted to a set of 1D data.

```
# Create some data
x = Double1d([3,4,6,7,8,10,11,13]) # These are the positions of the 1D data
y = Double1d([2,4,5,6,5,6,7,9]) # These are the data values at each position
# The created arrays are:
print x  # [3.0,4.0,6.0,7.0,8.0,10.0,11.0,13.0]
print y  # [2.0,4.0,5.0,6.0,5.0,6.0,7.0,9.0]

# Decide that we will fit it with a polynomial

model = PolynomialModel(3)

# The Fitter class expects the -'x' data point positions and the model.
# In the binomial case, a Double2d array of x,y values is required.
# The Fitter class deals with non-iterative models only.
# [Note: For non-linear models the fitter toolbox provides
# the AmoebaFitter and the LevenbergMarquardtFitter]

fitter = Fitter(x, model)

# Now we fit the data values(y); the returned array contains the parameters
# that make up a 3rd degree polynomial.
# Note: the model that we fed into the fitter is modified along the
# way, such that it contains the computed parameters of the polynomial.

poly = fitter.fit(y)

# Printing the fit results (truncate to 3 decimal places to fit in line)

print poly # [-6.921,4.463,-0.543,0.022]

# -..and also getting the Chi-squared. The fitter has already been applied
# and we can use the getChiSquared() method to determine the fit

print -"Chi-Squared = -", fitter.getChiSquared()
# Chi-Squared =  0.9933079890409999

# The fitted polynomial can then be applied as a function to interpolate
# between fitted points. Interpolate at -'n' uniformly spaced x values

n = 100
u = MIN(x) + Double1d.range(n + 1) * ((MAX(x) -- MIN(x)) -/ n)

# Apply the model
umodel = model(u)

# Now we can plot the data (x vs y) and the polynomial fit (u vs umodel)
# Set up the plot space
plot = PlotXY()
# Plot x against y in blue.
plot[0] = LayerXY(x, y, name = -"Data")
# Overlay a second plot showing the polynomial fit in green.
plot[1] = LayerXY(u, umodel, name = -"Fit", color = java.awt.Color.green)
```

Example 3.4. A 1D polynomial fit.

The final plotted display should look like Figure 3.2
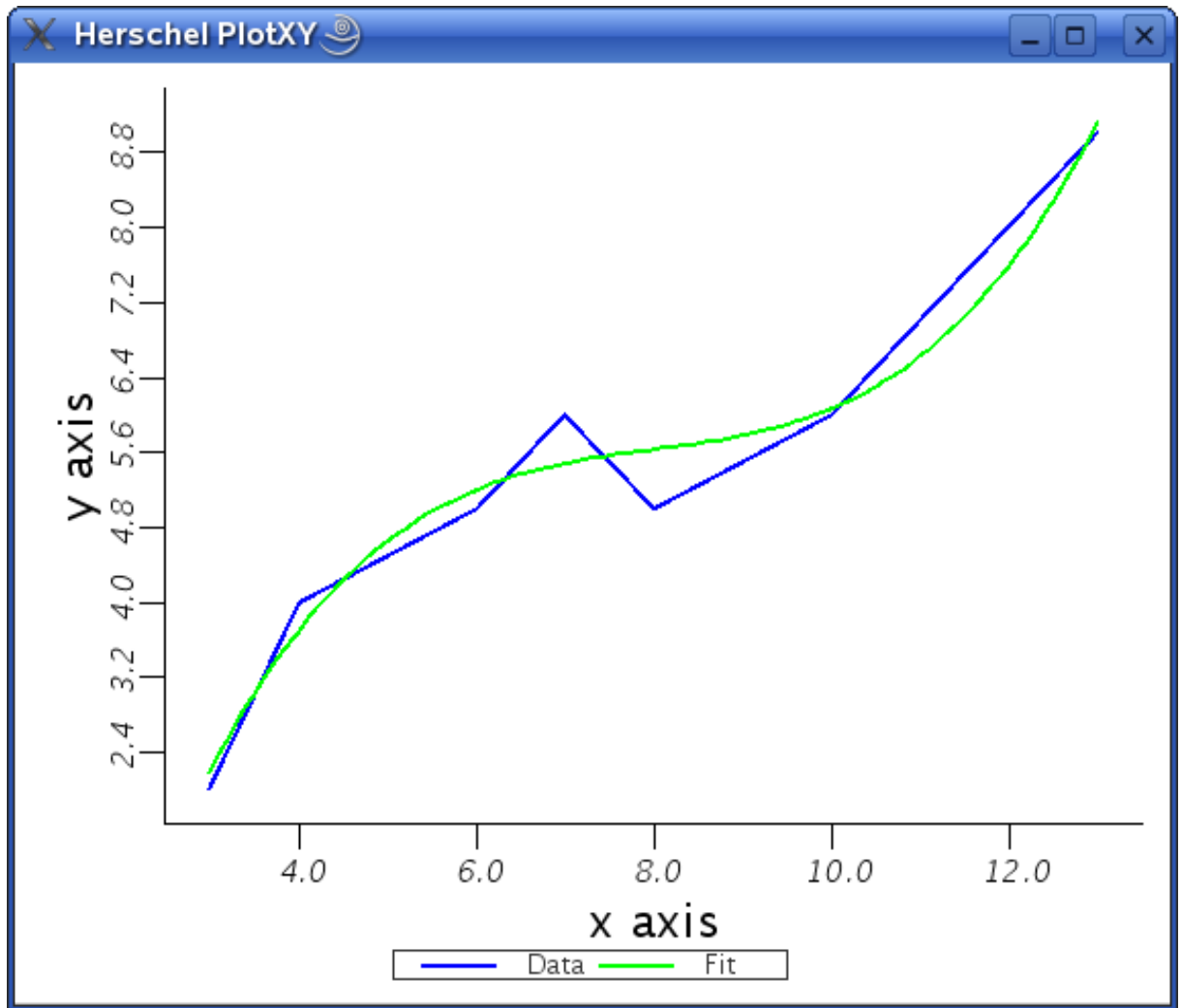
Figure 3.2. Illustration of polynomial fit.

2D fit example

For 2D data we express the positions at which we have data by a Double2d array -- this is basically a list of x, y positions at which we have known data values that we will fit a 2D Gaussian to. So the x array in our previous example is now replaced by a 2D array of data positions. The y array has the data values at those positions.

In Example 3.5, an array with values that provide a Gaussian with random noise added is fitted by the Gauss2D model.

```
# We start by making a little routine that creates the data for us.
# The output contains the -'xy' positions as a Double2d array and the data
# values are held in in the Double1d array -'y2'.
def makeData():
# Define some constants
        N = 9                  # We will create an array that is NxN
        a0 = 10.0              # Amplitude of gaussian
        x0 = 0.7              # x position of gaussian
        y0 = --0.3             # y position of gaussian
        s0 = 0.4             # Width
# Make data with an underlying gaussian model.
        x = Double1d.range(N) -/ 2.0 -- 2 # create x values
        NN = N * N # the number of x and y positions (NxN)
        xy = Double2d(NN, 2) # Create empty array of xy positions
        ym = Double1d(NN)    # Create empty array for amplitude of pure Gaussian
        y2 = Double1d(NN)    # Create empty array for Gaussian with noise (our
data).
# These have amplitude values only.
        rng = java.util.Random( 12345 -) #provide a random amplitude (noise)
# To add to our model Gaussian with a seed value.
        si = 1.0 -/ s0 #just inverse of Gaussian width to be used
        for i in Int1d.range(NN):
          xy[i,0] = x[i -/ N] # Fills x positions for our data array
          xy[i,1] = x[i % N] # Fills y positions for our data array
          xx = ( xy[i,0] -- x0 -) * si
          yy = ( xy[i,1] -- y0 -) * si
          ym[i] = a0 * EXP(-0.5 * xx * xx) * EXP(-0.5 * yy * yy)
          # Fills 1d array with amplitude values...
          y2[i] = ym[i] + 0.2 * rng.nextGaussian()  # -...and adds noise to it
        return xy,y2

# Create the array with a 2D gaussian in it using the above routine.
a = makeData()
# The first item in -"a" has the xy positions in it
xy=a[0]
# The second item has the data values
y2=a[1]

# Define the model to be used in the fit
gaus2d = Gauss2DModel()

# Define the fitter: LevenbergMarquardt, a non-linear fitter is needed for
# a gaussian fit. We could use an AmoebaFitter here also --- user preference.
fitter = LevenbergMarquardtFitter(xy, gaus2d)

# A useful way to make data formats prettier for the printout of our results
F = DataFormatter()
# Find the parameters
param = fitter.fit(y2)
print -"Parameters %s" % F.p(param)
# Parameters [ 9.645  0.694  --0.300 0.413]
print -"Parameters are:  gaussian height, x position, y position, width"
#Parameters are:  gaussian height, x position, y position, width
# Find the standard deviations of the all four parameters...
stdev = fitter.getStandardDeviation();
print -"Stand Devs %s" % F.p(stdev)
#Stand Devs [   0.218    0.009    0.009     0.007]

# -...and the chi-squared for the fit
print -"ChiSq      %s" % F.p(fitter.getChiSquared())
#ChiSq         3.552
```

Example 3.5. A 2D Gaussian fit

3.10.8. Spectral fitting

This section describes how to use the spectrum fitting toolbox in HCSS to fit a spectrum. To access the toolbox it will need to be loaded from the into the session. This can be done by typing in the following in the JIDE command line interface.

```
from herschel.ia.toolbox.spectrum.fit import *
```

The toolbox is continuing to be developed and it is expected that new features will be added to what is described here. Features that are certain to be added are listed in the 'To Be Added' section below.

3.10.8.1. Data format

The data that is used by the classes can be any Java or Jython object, as long as it implements the SpectralSegment interface (e.g., extracted from a Spectrum1d object).

You can create a SpectralSegment using a little helper class, `FitData`. This class takes two `Double1d`'s (representing wavelength/frequency and flux/values) and wraps them into a `SpectralSegment`.

If you have two Double1d arrays, x and y, then the statement:

```
data = FitData(x,y)
```

creates a `SpectralSegment`.

3.10.8.2. General usage

In general, data to be fitted contains three kinds of features:

- a background/continuum level

- >one or more spectral lines

- noise

These can be fitted using the `SpectrumFitter` tool.

The purpose of the `SpectrumFitter` is to fit models to the background and the spectral lines in such a way that when the models are subtracted from the data, the residual only contains the noise.

Although fitting spectral lines and the background does not differ mathematically, the two cases must be handled separately. That is, you better first fit the background, subtract that from the data, and only then fit the lines.

3.10.8.3. Fitting your data

As the user you interact with the `SpectrumFitter` tool. To have more control over the models (see below) you can also interact with the class `SpectrumModel`.

Note that you normally must know where (approximately) you expect a spectral feature in your data to be, plus its expected shape, and rough shape parameters. So, an initial guess is required - if this guess is completely wrong you may end-up fitting noise rather than your spectral lines.

The `SpectrumFitter` tool provides graphical information on the fitted data to assess the fits that are made.

3.10.8.4. A simple fit case

The simplest spectral fitting case involves data with one spectral line and with no background/continuum.

The basics are, a) create a `SpectrumFitter`; b) add models to it.

We assume you know that you have a `SpectralSegment` which contains the spectral line has a Gaussian shape that is located near x0, has an amplitude of about a0, and a width of about s0 (the exact values of a0 and s0 are not so important). The following is an example:

```
x = Double1d.range(15)
y = Double1d([0.0,0.1,-0.1,0.05,0.1,0.2,1.0,3.6, \
    2.5,1.5,0.7,0.0,0.1,-0.13,-0.01])
data=FitData(x,y)
# This has a peak near value number 7 with an
# amplitude of 3.6 and a width close to 1.
x0 = 7.0
a0 = 3.6
s0 = 1.0
# These are our initial guesses.
```

We can fit this using the `SpectrumFitter`:

```
sf = SpectrumFitter(data) # note that a plot of the data is
                          # automatically drawn in a separate window
# see Figure 3.3
sf.addModel('gauss', [a0, x0, s0]) # note the square brackets
                                   # and the order of the parameters
print sf        # this prints out the fitted Gaussian parameters
                # and their standard deviations.
# Fit results:
                # p0 = 3.3890821693817763, stddev= 0.2568383201833762
                # p1 = 7.444866152807009, stddev= 0.09308190130219554
                # p2 = 1.0796490360796016, stddev= 0.09333220808910589
# for the amplitude, position and width respectively.
```

Figure 3.3. Spectrum fit data setup.

The result of adding the model is the production of two further plots. One plot contains:

- the data (blue line)

- the input model as given by you (green line)

- the resulting fit (red line)

The second plot displays the residuals. See Figure 3.4 and Figure 3.5.

Figure 3.4. Data fit - data in blue, input model in green, fit in red

Figure 3.5. Residuals on the fitted data

3.10.8.5. Available models for fitting

There are a number of models available for fitting. In order to see the available models in the system at any time you can use the following.

```
print SpectrumFitter().info()
```

This command provides a listing of available models that can be fit. If we pick one of these models we can get more information on it. For example we can look to fit a polynomial -- the 'poly' model.

```
print SpectrumFitter().info('poly')
```

This indicates there is one constructor (only one way of calling it). The order needs to be given in one array and initial parameter guesses in a second array.

```
from herschel.ia.toolbox.spectrum.fit import *
from herschel.ia.toolbox.spectrum.fit.testdata import *
#There are 7 inbuilt datasets for spectrum fit checking and illustration
m = MakeData(3)  # integer value represents different models
m.addNoise(10) # add some noise to the data
# now do fit --- the guess and final model fit are displayed overlayed on the data
sf = SpectrumFitter(m) # setup spectrumfitter
mod=sf.addModel('poly',[3],[1.0,0.0,0.0,0.0])
# 3rd order poly model and guess for fit parameters
sf.doFit()  # fit displayed.
print sf # provides fitted parameters with their standard deviations
```

The models currently available and an illustration of their use is given in Table 3.1.

Table 3.1. Spectrum fit model types and their use.

Name	Example use -- names in brackets should be replaced by numerical values representing the initial guess for the parameter(s)
'atan'	mod=sf.addModel('atan',[amplitude,slope,offset])
'exp'	mod=sf.addModel('exp',[amplitude,exponent])
'gauss'	mod=sf.addModel('gauss',[amplitude, position, width])
'gaussmix'	mod=sf.addModel('gaussmix',[amplitude, position, width])
'harmonic'	mod=sf.addModel('harmonic',[Order,Period],[params]). Number of parameters provided = 2*order + 1
'lorentz'	mod=sf.addModel('lorentz',[amplitude, shift, gamma])
'pade'	mod=sf.addModel('pade',[Num,Denom],[params]). Number of parameters provided = Num + Denom + 1
'poly'	mod=sf.addModel('poly',[Order],[params]). Number of parameters provided = Order + 1
'power'	mod=sf.addModel('power',[Degree],[param]). Number of parameters provided = 1
'sinc'	mod=sf.addModel('sinc',[amplitude, position, width])
'sine'	mod=sf.addModel('sine',[frequency, cosine amp, sine amp])
'sineamp'	mod=sf.addModel('sineamp',[frequency], [two params])
'sinemixed'	mod=sf.addModel('sinemixed', [frequency, cosine amp, sine amp])

3.10.8.6. Multiple line fitting

If, in the simple line case above, the residual is only noise, you have completed your fit. If not, then there may be another spectral line in your data. From the original data or from the residual you can

often determine the initial parameters of a second line: a1, x1, s1. In order to include a fit to this second line also we can simply add another model to the fitter by using the 'addModel' method:

```
sf.addModel('gauss', [a1, x1, s1])
```

This will update the fit and plots automatically. In the first plot you will now also see the two models separately using the fitted parameters as black lines.

3.10.8.7. Background/continuum fitting

Background/continuum fitting is not treated differently from the above. The only difference is the model used to fit the background.

When being combined with spectral line fits, it is best to fit the background first then add the spectral line model fit. If you don't, the fit of your spectral lines will initially be quite poor.

One model to use for a background is a polynomial. For a first order Polynomial (y = c0 + c1*x):

```
sf.addModel('poly', 1, [c0, c1]) # the second value is the polynomial order
```

For a higher order (n):

```
sf.addModel('poly', n, [c0, c1, -..., cn])
```

3.10.8.8. Fit of line and continuum

We can fit a line and continuum simultaneously by adding more than one model before doing the fit (e.g., a polynomial and gaussian model). We can then do a global fit. An example is given below.

```
#import the appropriate packages
from herschel.ia.toolbox.spectrum.fit import *
from herschel.ia.toolbox.spectrum.fit.testdata import *
m = MakeData(5)  # values represent different models
m.addNoise(10) # add noise to the model
sf = SpectrumFitter(m)
mod=sf.addModel('gauss',[4.0,30.0,1.0]) #also plots initial guess
mod=sf.addModel('gauss',[1.2,10.0,2.0]) #second line
mod=sf.addModel('poly',[3.0],[0.0,0.0,0.0,0.0]) # polynomial for continuum
sf.doGlobalFit()  #fits all models at the same time --- residual plot also shown
```

The results of this are a plot of the data, initial guess and fit (in red) plus a separate plot of the fit residuals (see Figure 3.6 and Figure 3.7).
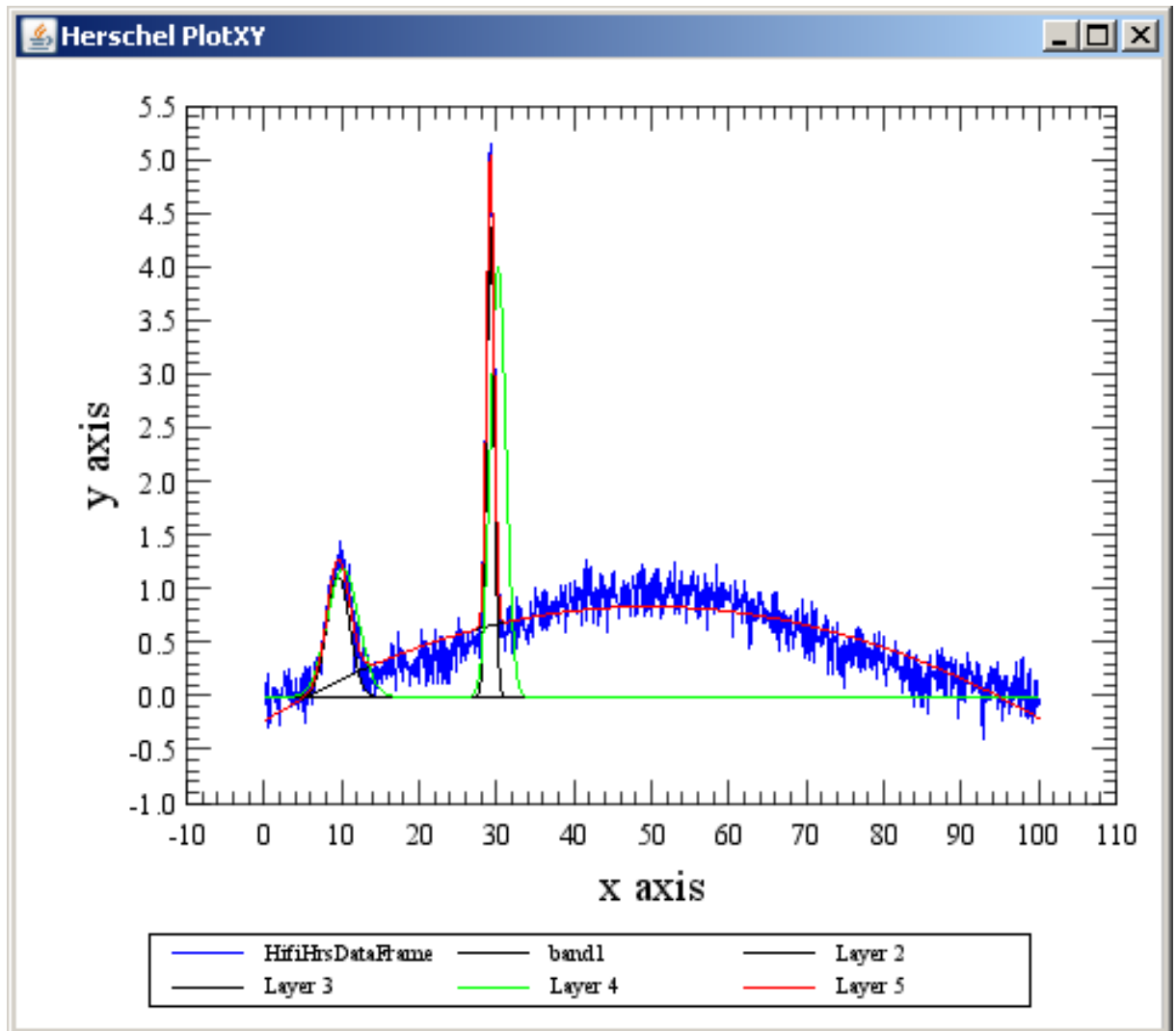
Figure 3.6. Fit using multiple models. In black are the individual guesses, in green the total initial guess and in red the final fit.
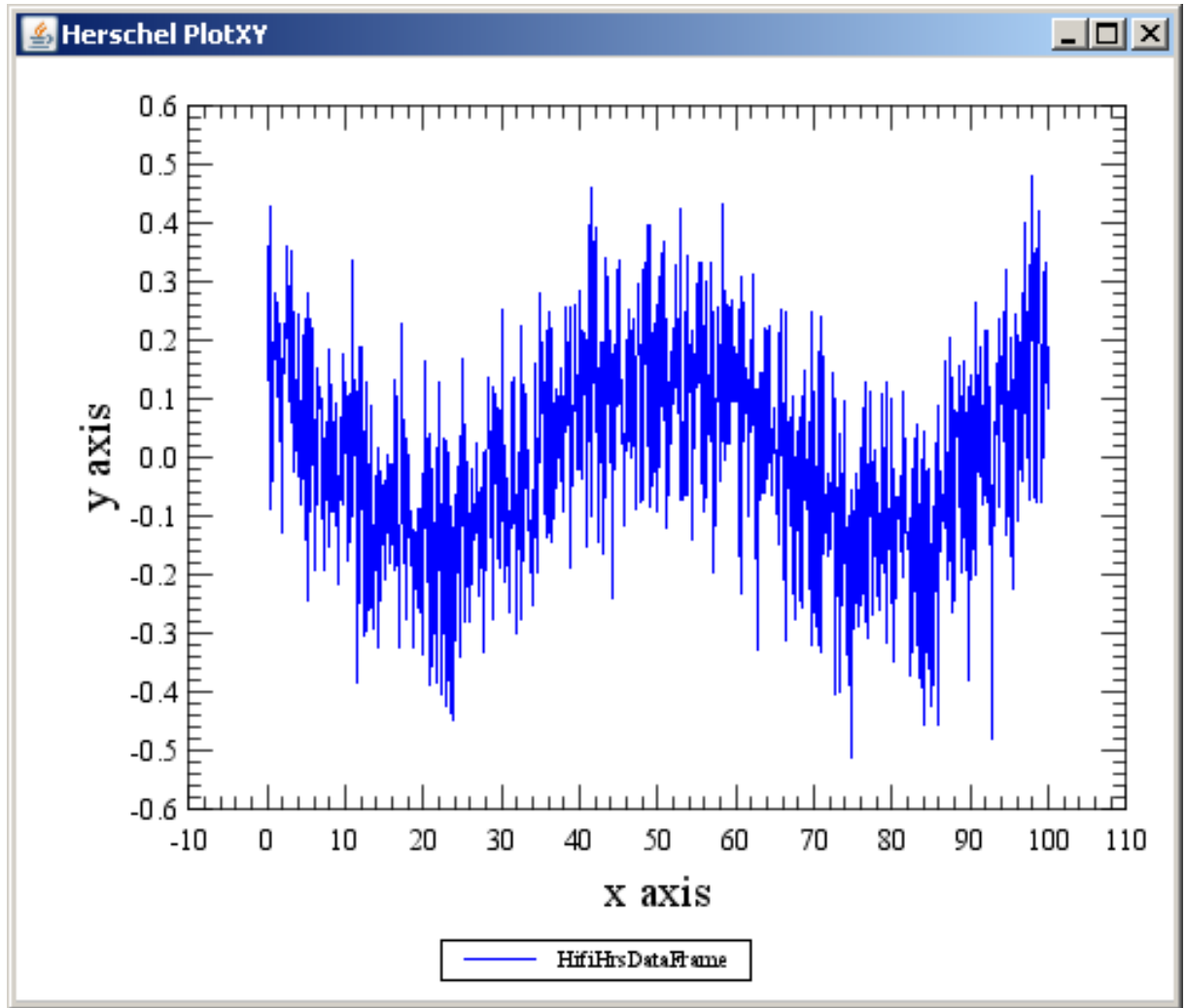
Figure 3.7. Residuals on the multiple model fit data shown in Figure 3.6.

3.10.8.9. Changing parameters

If you wish to change the initial parameters of any of the models, you can use the `setParameters` method of the models. To use them you must have a reference (label) to the model. This is in fact the return value of the `addModel` operation. In the example below, the label is simply 'm':

```
m = sf.addModel(...) # m is now a reference we can do things with
```

To change the initial parameters of the model

```
m.setParameters([...])
```

A new fit will be made on the fly and your plot display updated.

3.10.8.10. Removing fitted models

Removing models can only be done when you have a reference to the Model (as above). There are two ways to remove models:

```
sf.removeModel(m)
```

Or:

```
m.remove()
```

3.10.8.11. Using fit parameters

Once you are satisfied with a fit, you can set the fitted parameters as the default for the models:

```
m.useResults()
```

This may be useful when using the same models for a following dataset.

3.10.8.12. Subtracting a fit

You can subtract the model from the dataset:

```
sf.subtractModel(m)
```

This also removes the model from the fitter tool.

3.10.8.13. New data

Once you are satisfied with your models, you may want to apply them to a different dataset as well. This can be done with the operation:

```
sf.setData(otherData) # this replaces the data held in the
                      # SpectrumFitter with the SpectralSegment
                      # held in the variable -'otherData'
```

Once again, the fit will be redone on the fly.

3.10.8.14. Functions to be added in the future

>The following features are likely to be added to the system:

- add more model types

- subtract a model from the data, continue with the residuals;

- fix any of the given parameters in a model;

- select parts of the X-axis to include/exclude in the fit;

- make an initial guess for the model parameters.

3.10.9. Masks

The Numeric library offers two classes for handling data masks:

- `FixedMask` represents a *traditional* mask definition, with different masks (up to 64) defined at different bit offset positions. Note that this class only stores mask *definitions*, with mask data stored in different arrays. For more information and several examples, see the entry in the *User's Reference Manual*: [Section 2.132](#).

- `PackedMask` instead stores the mask data itself. There is in principle no limit on the number of masks that can be stored in a single `PackedMask` object. For more information and several examples, see the entry in the *User's Reference Manual*: [Section 2.276](#).

3.10.10. Matrices

Most of the utilities for dealing with matrices are provided by the numeric.toolbox.matrix package. However, we must not forget that simple vectors are just matrices with just one row (or one column),

so even vector classes like `Double1d` provides tools like a `dotProduct` method for scalar multiplication of vectors:

```
x = Double1d([1,2,3,4])
y = Double1d([1,3,5,7])
print x.dotProduct(y) # 50.0
```

We now take a closer look at the numeric.toolbox.matrix package and its classes and function objects for matrix manipulation.

Transpose

To transpose a matrix do the following:

```
A = Int2d([[1,2],[3,4],[5,6]])
print TRANSPOSE(A) # [[1,3,5],[2,4,6]]
```

Determinant

Use this function to find the determinant of a square matrix given by a `Double2d` array.

```
A = Double2d([[1,2],[3,4]])
print DETERMINANT(A) # --2.0
```

Note: This currently does not work for complex matrices.

Inverse

You can find the inverse of a square matrix as follows:

```
A = Float2d([[1,2],[3,4]])
print INVERSE(A) # [[-2.0,1.0],[1.5,-0.5]]
```

Note: This currently does not work for complex matrices.

Matrix multiplication

Use `MatrixMultiply` for matrix multiplication:

```
x = Double2d([[2,4,6],[1,3,5]])
y = TRANSPOSE(x)
z = MatrixMultiply(y)(x)
print z
```

It is important **not** to use the Jython * operator for matrix multiplication. However, the + operator performs element-wise addition as expected.

It is also possible to multiply a matrix by a vector as follows (since, as we already pointed out, a vector is nothing more than a matrix with just one row or column):

```
a = Double2d([[1,2,3],[7,5,4],[7,4,9]])
b = Double1d([4,1,7])
print MatrixMultiply(b)(a) # [27.0,61.0,95.0]
```

Warning

The correct syntax to multiply matrix a by matrix b is `MatrixMultiply(b)(a)`.

LU decomposition

For an *m x n* matrix *A*, LU decomposition returns matrices *P*, *L* and *U* so that *PA = LU*:

- *P* is a *permutation matrix*, so that the product *PA* results in a permutation of *A*'s rows. In the class described below, *P* is replaced by an equivalent *permutation vector* ***p***.

- *L* is a unit lower triangular matrix.

- *U* is an upper triangular matrix.

The `LUDecomposition` class provides this functionality. The following example shows how it is used:

```
A = Double2d( [ [1,1,1],[1,2,3],[1,3,6] -] -)
print A
# [
# [1.0,1.0,1.0],
# [1.0,2.0,3.0],
# [1.0,3.0,6.0]
# -]
d = LUDecomposition(A)
print d.l # Getting L
# [
# [1.0,0.0,0.0],
# [1.0,1.0,0.0],
# [1.0,0.5,1.0]
# -]
print d.u # Getting U
# [
# [1.0,1.0,1.0],
# [0.0,2.0,5.0],
# [0.0,0.0,-0.5]
# -]
print d.pivot # Getting the permutation vector
# [0,2,1]
```

You can easily verify that the result is correct:

```
print MatrixMultiply(d.u)(d.l)
# [
# [1.0,1.0,1.0],
# [1.0,3.0,6.0],
# [1.0,2.0,3.0]
# -]
```

LU gives *A* with the row order changed as described by the permutation vector: row 0, then row 2, then row 1.

Eigenvalue decomposition

The `EigenvalueDecomposition` class provides eigenvalues and eigenvectors of a real matrix. The following examples shows how it can be used:

```
A = Double2d( [[1,1,1],[1,2,3],[1,3,6]] -)  # Creating matrix
evd = A.apply(EigenvalueDecomposition())   # Performing decomposition
D = evd.d  # Obtaining the block diagonal eigenvalue matrix
V = evd.v  # Obtaining the eigenvector matrix
print evd.imagEigenvalues  # Printing the imaginary parts of the eigenvalues
print evd.realEigenvalues  # Printing the real parts of the eigenvalues
print evd.vcond  # Printing the condition (2-norm) of the matrix, defined as
                 the ratio of the highest and smallest singular value
```

If *A* is symmetric, then $A = V D V^{T}$, where the eigenvalue matrix *D* is diagonal and the eigenvector matrix *V* is orthogonal.

If *A* is not symmetric, then the eigenvalue matrix *D* is block diagonal with the real eigenvalues in 1-by-1 blocks and any complex eigenvalues, $\lambda + i \mu$, in 2-by-2 blocks, $[\lambda, \mu; -\mu, \lambda]$. The columns of *V* represent the eigenvectors in the sense that $A V = V D$. The matrix *V* may be badly conditioned, or even singular, so the validity of the equation $A = V D V^{-1}$ depends upon `vcond`.

Singular value decomposition

For an *m x n* matrix *A* with *m >= n*, the singular value decomposition is an *m x n* orthogonal matrix *U*, an *n x n* diagonal matrix *S*, and an *n x n* orthogonal matrix *V* so that $A = U S V'$.

The singular values, $\sigma_k = S_{kk}$, are ordered so that $\sigma_0 >= \sigma_1 >= ... >= \sigma_{n-1}$.

The singular value decomposition always exists, so the constructor will never fail. The matrix condition number and the effective numerical rank can be computed from this decomposition.

The `SingularValueDecomposition` class provides this functionality. For more information see the *User Reference Manual*: Section 2.367.

Matrix equations

Use `MatrixSolve` to solve matrix equations. For example, if you wanted to solve the matrix equation: `A.X = B`:

```
x = MatrixSolve(b)(a)
print x  # [-0.9838709677419352,0.5322580645161287,1.3064516129032258]
```

A note on naming conventions

You might find a bit confusing that some names, like `dotProduct`, start with a lowercase letter and have all the other initials capitalised, while other names, like `MatrixMultiply`, have *all* initials capitalised, and there is a fair share of names like `TRANSPOSE` with all uppercase letters. You can find more about these quirks in the appropriately named Section 1.19.

3.10.11. Random numbers

Note

For simplicity we will speak of *random* numbers throughout this section, even if we know very well that a computer can only create *pseudo*random numbers. Discussing the subtleties of generating (pseudo-)random numbers on a computer is beyond the scope of this section.

To create random numbers with DP we first have to instantiate a *generator*. There are three generators currently available:

- `RandomUniform`: generates random numbers in the range $0 <= x < 1$ if invoked without parameters, like this:

```
myGenerator = RandomUniform()
```

It is also possible to give a maximum value different from 1 to have random numbers created in the range $0 <= x < max$:

```
myGenerator = RandomUniform(max)
```

- `RandomGauss`: generates random numbers following a Gaussian distribution.

- `RandomPoisson`: generates random numbers following a Poisson distribution of specified mean value greater than zero. It is instantiated like this:

```
myGenerator = RandomPoisson(mean)
```

It can only produce integer-type random numbers (`int`, `short` and `long`).

In all cases what is being used under the hood is the Donald Knuth generator (see *The Art of Computer Programming*, Volume 2, Section 3.2.1) as implemented in the `java.util.Random` class.

Once we have a generator in place, how do we create random numbers? The handy feature is that we can create a single scalar random number or an array of any size and dimension we like (as long as it fits in memory). Just put the type of numeric value you want as input, and the output will be the same thing, but populated with random numbers. A few examples:

```
myGenerator = RandomUniform()    # Generating random numbers between 0 and 1
print myGenerator(0.0)           # We want a floating point random number...
# 0.8754230073094597             # -...and there it is (don't expect to get the
                                 # same number)
x = Double1d(10)                 # Now for an array of ten doubles...
print myGenerator(x)             # We leave it to you to see the result
print myGenerator(Double1d(10))  # Of course you can create the input on the fly
print myGenerator(Int1d(100))    # What's the result of this one? Does it make sense?
```

You might have been puzzled to see a hundred zeroes scroll on your screen after executing the last command of the example. It's not so surprising if we think that we asked the computer to produce *integer* random numbers between zero and one, *excluding one*. The choice of possible values was pretty limited.

If we want to change the *seed* of the random number generator we can do so by the `setSeed` method, which takes a long parameter as an input:

```
myGenerator.setSeed(54653856L)
```

3.10.12. Numeric integration

Numeric integration in DP is implemented via an *Integrator* interface. The function to be integrated has to be declared as a class of a *RealFunction* containing a method called *calc* which takes one argument, the independent variable.

The following Integrators for a standard integration interval [a,b] are available:

- RectangularIntegrator

- RombergIntegrator

- SimpsonIntegrator

- TrapezoidalIntegrator

- GaussianQuad4Integrator

- GaussianQuad5Integrator

- GaussLegendreIntegrator

All these integrators have two arguments for initialisation: the lower limit of integration (a) and the upper limit (b). Once the integrator is initialised and the user function is defined then to perform the integration a method called *integrate()* is executed with an argument the user function. This is shown in the following example:

```
from herschel.ia.numeric.toolbox import RealFunction
```

```
class MyFunction(RealFunction):
  def calc(self,x):
    return x*x

f = MyFunction()
a = --3.0
b = 3.0
i = RombergIntegrator(a, b)
print i.integrate(f) # 18.0
print -"Analytical answer: -",(b**3 -- a**3)/3.0
```

The following special cases of numeric integration are also implemented:

- GaussHermiteIntegrator: for integration with limits (-Inf,+Inf) of a special class of functions

$$\int_{-\infty}^{+\infty} e^{-x^2} f(x)\, dx$$

- GaussLaguerreIntegrator: for integration with limits [0,+Inf] of a special class of functions

$$\int_{0}^{+\infty} x^{\alpha} e^{-x} f(x)\, dx$$

The input for the integrator initialisation is α.

- GaussJacobiIntegrator: for integration with limits [-1,1] for a special class of functions

$$\int_{-1}^{1} (1-x)^{\alpha} (1+x)^{\beta} f(x)\, dx$$

The input for the integrator initialisation are α and β.

If a tabular data of x,y is to be integrated then it is necessary to interpolate first and then apply a suitable integrator. This is shown in the following example:

```
from herschel.ia.numeric.toolbox import RealFunction

x = 0.1 + 1.9*Double1d.range(11)/10.0 # 11 points between 0.1 and 2.0
y = 1.0/x

f = CubicSplineInterpolator(x,y) # interpolate first.
a = 0.1
b = 2.0
integrator = SimpsonIntegrator(a, b) # use Simpson's rule

res = integrator.integrate(f) #
print -"Result: -",res
print -"Analytical result: -",LOG(b) -- LOG(a)
```

3.10.13. Interpolating discrete data

If the objective is to integrate discrete data, this can be done by means of a `FitterFunction`, which is a function that interpolates the given data, with a specific model. For example:

```
from herschel.ia.toolbox.fit import FitterFunction

# x, y are Double1d that represent the abscissas and values of our data
```

```
f = FitterFunction(x, y, PolynomialModel(3))                    # Uses a Fitter
g = FitterFunction(x, y, PolynomialModel(2), FitterFunction.AMOEBA)
# Uses an AmoebaFitter
```

If more precise fitting is needed, you can do it by yourself, and then pass the already built fitter (or the model) to the FitterFunction:

```
# x, y are Double1d that represent the abscissas and values of the data
model = CubicSplinesModel(x)
fitter = AmoebaFitter(x, model)
fitter.setSimplex(params, range) # customize the fitter as you want
fitter.fit(y)
f = FitterFunction(fitter) # or f = FitterFunction(model)
```

If one of the defined interpolators suites your needs, it can be used directly, instead of a `FitterFunction`. For example:

```
# x, y are Double1d that represent the abscissas and values of the data
f = CubicSplineInterpolator(x, y)
```

3.11. Example programs

The HCSS distribution includes a number of Jython example programs that demonstrate not only basic arrays functions but also use of filters, fitters, Fourier transforms, etc. They are currently kept at ftp://ftp.rssd.esa.int/pub/HERSCHEL/csdt/releases/doc_ia/ia/demo/scripts. These are:

numeric_whatisnew.py	Example of the newest components of the numeric package.
numeric_demo.py	Example of how to use the 1D functionality.
numeric_2D_demo.py	Example of how to use the 2D functionality
convolution_demo.py	Example of how to use the convolution functionality
polyfitter_demo.py	Example of how to perform polynomial fitting

3.12. Mathematical operations on spectra

3.12.1. Introduction

The spectrum arithmetic toolbox allows to combine Herschel spectrum data. Operations are performed either on subclasses of spectrum datasets (`Spectrum1d`, `Spectrum2d`), on cubes (`SimpleCube`, `SlicedCube`), or on products containing such data structures (e.g., HifiTimelineProduct).

Operations on Spectra include Selection and Arithmetic Operations.

- *Selection*: Provide means of selecting those spectra that can be combined. For instance HIFI cold-load spectra, ON spectra, etc. Selection can be applied to datasets, such as rows of a `Spectrum2d`, or to tables within a product, such as datasets included in a HifiTimelineProduct.

- *Arithmetic Operations*: Provide means of combining the selected spectra. This includes:

  - Basic arithmetic operations such as addition, subtraction, multiplication, or applications of scalar functions.

  - Statistical operations such as mean, median, variance, standard deviation or percentiles for samples / selections of spectra.

- Data transformations such as smoothing or frequency re-sampling.

It is planned that the arithmetic toolbox will provide generic functionality for all instruments (HIFI, PACS and SPIRE). Instrument-specific behaviour will be pre-configured by defaults in the system but can also be overwritten by the user.

3.12.2. Toolbox primer: selection

We present the power of the toolbox with a few code examples. Assume we have started a jide session and loaded a `Spectrum2d` dataset with name 'data' from a local pool or a database.

We might want to work only with a sub-set of the spectra included in our data. For a `Spectrum2d` this means we have to (1) select specific rows from the data and (2) combine them into a new dataset by applying some arithmetic operations on the selection. Task (1) is performed with the `SelectSpectrum` task,

```
from herschel.ia.toolbox.spectrum import SelectSpectrum
```

The SelectSpectrum-task can be configured and used in many different ways. A frequent usage is to identify all the rows of the dataset that have a specific value in a particular column:

```
ds1 = SelectSpectrum()(ds=data, selection_lookup={"bbtype":[3260]})
```

The example above selects all the rows with a value=3260 in the column named 'bbtype'. Hence, the selection is performed by using the keyword selection_lookup in the call of the task, using what is called a *python dictionary*. This py-dictionary contains the name of the attribute to look up as key (column name) and the attribute value as value. All the rows in the resulting dataset ds1 have values 3514 in the bbtype column.

Using py-dictionaries suggests that we may combine several selections by adding further lookup properties to the dictionary. Indeed, all the rows in the dataset resulting from

```
ds1 = select(ds=data, selection_lookup={"bbtype":[3260],"buffer":[1]})
```

```
ds2 = select(ds=data, selection_lookup={"bbtype":[3260],"buffer":[2]})
```

have values 3260 in the bbtype column and values 1 in the buffer column (hence ds2 is a subset of ds1). Note that the lookup values are specified as py-lists. By specifying a list of admissible values those spectra are selected that match one of values found in the list. As will be explained below, there are other selection models better suited for floating point values.

3.12.2.1. More on selection methods

- Lookup specific attribute value(s):

  For one (or several) discrete criteria use the keyword selection_lookup:

  ```
  ds1 = select(ds=data, selection_lookup={"bbtype":[3413]})
  ```

  Spectra with bbtype=3413 are selected and included in the result container.

  ```
  ds2 = select(ds=data, selection_lookup={"bbtype":[3412, 3413]})
  ```

  Spectra with bbtype=3412 or bbtype=3413 are selected and included in the result container.

  ```
  ds3 = select(ds=data, selection_lookup={"bbtype":[3413],"buffer":[1]})
  ```

Spectra with bbtype=3413 and buffer=1 are selected and included in the result container.

- Index selection:

  If you want to select specific spectra included in the container by its index, use the keyword selection_index:

  ```
  ds1 = select(ds=data, selection_index=[1,5,12])
  ```

  The spectra with indices 1, 5, 12 are selected and included in the result container.

- More general selection model:

  Use the keyword selection and use one of the selection models found in the package

  ```
  herschel.ia.toolbox.spectrum.selections.models
  chopperSelection = RangesSelectionModel("Chopper", [-4.4, 5.9], 0.1)
  ```

  The first parameter specifies the name of the attribute, the second parameter gives an array of centers of the ranges and with the third parameter you specify the radius of the ranges to be considered. In summary, this ranges selection model will identify all spectra for which the attribute "Chopper" has values located within a distance $r = 0.1$ around one of the centers [$z1=-4.4,z2=5.9$].

  ```
  ds4 = select(ds=data, selection=chopperSelection)
  ```

  For further selection models see further down in the documentation.

3.12.3. Toolbox primer: average spectra

After selecting the data, we can move to task (2), the application of some arithmetic operations to the selected spectra. For example, if we now want to average the selection, we can invoke the `AverageSpectrum` task:

```
from herschel.ia.toolbox.spectrum import AverageSpectrum

avg21 = AverageSpectrum()(ds=ds2)
```

The selection explained in task (1) can also be included in the average spectrum task, thus allowing to perform selection and averaging in one step:

```
avg22 = AverageSpectrum()(ds=data, selection_lookup={"bbtype":[3260],"buffer":[2]})
```

This result is identical to the separate operations. It includes a single row with the average flux. The resulting dataset contains exactly the same columns as the input dataset. Thus, what values should we fill in the columns not affected by the operation? This is determined by a default action that depends on the input data type (sub-class of `Spectrum2d` in our example). For the `Spectrum2d`, the default action consists of copying the values found in the input spectrum.

This way of processing the data is general: we always try to keep as much information as possible. All columns and also the meta data are set in a type specific, instrument specific, or user specific way. The output data type is the same as the input data type.

The toolbox operations are not restricted to operations on Spectrum2d as our example may suggest. In all the operations in the herschel.ia.toolbox.spectrum no reference is made to Spectrum2d. The operations only refer to a specific contract (a java-interface), the SpectrumContainer-interface. Spectrum2d also fulfills this contract. All the datastructures that obey this contract can be processed by the arithmetic tools. The efforts to have this contract implemented for other data types is relatively small.

3.12.4. Toolbox primer: subtract spectra

Other arithmetic operations are available such as pair operations (subtract, divide, pair-wise add/ multiply) and scalar operations (add/subtract or multiply/divide by a scalar quantity). Here is an example that shows how to use the subtraction:

```
from herschel.ia.toolbox.spectrum import SubtractSpectrum

diff12 = SubtractSpectrum()(ds1=ds1, ds2=ds2)
```

Here, the datasets ds1 and ds2 either must have the same number of rows, or one of them must have only a single row. If they have the same number of rows, the subtraction is carried through for the flux data on a row-by-row basis. If the second contains only one row, this row is subtracted from all the rows in the first dataset (or the other way around).

The same task can also be used for subtracting a scalar:

```
ds_m2= SubtractSpectrum()(ds=data, param=2.0)
```

Here the number two is subtracted from all the flux columns in our data.

3.12.5. Toolbox primer: divide spectra

The use of the `DivideSpectrum` -task is identical:

```
from herschel.ia.toolbox.spectrum import DivideSpectrum

ratio12 = DivideSpectrum()(ds1=ds1, ds2=ds2)
ds_d2 = DivideSpectrum()(ds=data,param=2)
```

3.12.6. Toolbox primer: add and muliply spectra

Similarly, for multiplication and addition we can import tasks that can be used in a similar fashion.

```
from herschel.ia.toolbox.spectrum import MultiplySpectrum
from herschel.ia.toolbox.spectrum import AddSpectrum
```

These tasks work in exactly the same way.

3.12.7. Toolbox primer: resample and smooth spectra

Additional tasks included in the toolbox include smoothing, frequency resampling or extracting/ cutting the spectra. The system again provides the instance

```
from herschel.ia.toolbox.spectrum import ReamplingFrequency

resample = ReamplingFrequency()
```

which allows for resampling non-equidistant grids to linear grids and the other way around. Resampling to a linear grid with given resolution (width) would look like

```
data_resampled = resample(ds=data, density=true, resolution=1.0)
```

where the resolution is given in the same units as the frequencies in the data. The density parameter indicates whether the flux is specified as a per channel (true) or as a per frequency unit quantity (false).

For the smoothing, the instance

```
from herschel.ia.toolbox.spectrum import SmoothSpectrum

smooth = SmoothSpectrum()
```

is again loaded automatically by the system and it can be used by

```
data_smoothed = smooth(ds=data, filter="box", width=10)
```

3.12.8. Toolbox primer: statistics on spectra

Finally, the toolbox also allows to compute the statistics for the spectra included in a spectrum container.

```
from herschel.ia.toolbox.spectrum import SpectrumStatistics

statistics = SpectrumStatistics()
```

There are two alternative ways to compute the statistics for the spectra included in a spectrum container, the statistics computed on a per channel basis over all the spectra included in the container, or the statistics computed for each spectrum included in the container across the channels, possibly restricted to a range.

```
stats = statistics(ds=data)
```

The result of this operation stats is a product which contains the per channel statistics in Spectrum1d and the across channel statistics in a suitable TableDataset.

3.12.9. Summary of toolbox operations

Operations are available both at the task level and at the java level. The tasks are most suited for being used from the command line. The java classes which are wrapped by the tasks might be more helpful when developers want to integrate the functionality into other modules. The java classes will be discussed in the developer's sections.

- *SelectSpectrum (use select)*: Select spectra from a container and create a new spectrum container of the same runtime type.

- *AverageSpectrum (use avg)*: Average the spectra included in the container on a channel by channel basis. Restrict the average to specific selections or define groups and apply the average on a per group basis.

- *AddSpectrum (use add)*: Pairwise or scalar add.

- *SubtractSpectrum (use subtract)*: Pairwise or scalar subtract.

- *DivideSpectrum (use divide)*. Pairwise or scalar divide.

- *MultiplySpectrum (use multiply)*: Pairwise or scalar multiply.

- *ResampleFrequency (use resample)*: Resample each spectrum included in the container to a new, not necessarily linear grid.

- *SmoothSpectrum (use smooth)*: Smooth each spectrum included in the container.

- *ExtractFreqRanges (use extract)*: Cut the spectra included in the container to given frequency intervals.

- *ReplaceFreqRanges (use replace)*: Replace spectrum information in one container by information from another.

- *SpectrumStatistics (use statistics)*: Compute statistics of the spectra in the container - either on a per channel basis or across the channels.

3.12.9.1. Remarks

1. Fitting: There is a separate documentation on fitting: see the module ...

2. Datastructures: As indicate in the primer, all the data structures that fulfill the contract a spectrum container must have can be processed by the toolbox modules. Currently:

   - Spectrum1d: implements contract.

   - Spectrum2d: implements contract.

   - Cubes: under consideration.

   - Other instrument-specific data structures (such as HifiTimelineProduct or SpectrometerDetectorSpectrum): under consideration.

Chapter 4. Introduction to Tasks

This chapter aims to be an introduction for users to the Task framework. Writing Tasks allows us to create modular and reusable code for data reduction and analysis, easier to distribute and to be used by people other than the author.

4.1. The Task framework

When we were talking about OOP in Chapter 1, we used as example a very real and tangible object like an airplane. However, we mentioned that objects can also represent more abstract concepts. Dealing with astronomical data presents us with such a situation. When reducing or otherwise treating our data we go through a succession of self-contained operations. Data enter each of these "boxes" in a certain state and exit in a modified state. We might want to have a general template to represent such boxes, with a way to specify input and output parameters and check for their consistency. It would also be great to have some form of history to track what we have been doing to a given set of data, without the need to write it in a separate place or try and squeeze the information in the file name. Another handy tool would be a command to get help on that particular "box", to know at a glance what it does and what kind of parameter it expects.

The `Task` framework provides it all. Here we can see many concepts of OOP in action: reusable code (that of the `Task` class) to create modular pieces of software (our tasks) easy to plug together into more complex structures. In the following sections we will learn how to write a Task in Jython.

4.2. My first Task

4.2.1. Before the Task

Before writing a Task we should have something to turn into a Task. Paste the following code into your HIPE Editor view and then execute it with the double arrow button in the HIPE toolbar.

```
#-------------------------------------------------
# Average function
# Takes a TableDataset as input
# Returns a Double1d (1D array of real numbers)
# in which each row is the average of the values
# in the input table columns
#-------------------------------------------------
# Routine for calculating the average
def average(table):
    columns = table.columnCount
    divider = 1.0 -/ columns
    result = Double1d(table.rowCount)
    for column in Int1d.range(columns):
        result.add(table.getColumn(column).data)
    return result.multiply(divider)

# Routine for creating the initial table
def createTable():
    # Create array x (0.0, 1.0, 2.0, 3.0, 4.0)
    x = Double1d.range(5)
    columns = 5
    # Create an empty table with a name
    table = TableDataset(description = -"A test table")
    # Iterate for the the number of columns to fill up the table
    # Using -" -"%i" % column -" creates a string name for the
    # table-column which contains the integer value contained in
    # the variable name that appears after -"%". In this case
    # column labels are just 0 1 2 3 4.
    for column in Int1d.range(columns):
        table["%i" % column] = Column(x)
        x = x + 1
    # Return the result, a table called -'table'
    return table

# Routine for checking it out!!
def testAverage():
    # Create the table
    table = createTable()
    # Get the average and put it into an array called -'result'
    result = average(table)
    # Print the result (a 1D array)
    print -'Result:', result
```

Example 4.1. Before the Task

The above code has three functions in it. The important one is average, which does the "useful" bit of computation, giving the average of each column of a TableDataset. The createTable function simply creates the input TableDataset for average, while testAverage just calls the two functions above and prints out the result.

You can see how the above works by the following. The brackets indicate it is a function.

```
testAverage()   # Result: [2.0,3.0,4.0,5.0,6.0]
```

4.2.2. What makes a Task?

In the current implementation, a task has two components:

- *Signature.* Someone's signature is something by which we can unambiguously identify that person (leaving forgery aside). In the same way a Task's signature, consisting of its name *and* the number and type of input parameters, is a way to identify the Task with no ambiguity.

- *Execution.* This component is made of three *methods*, i.e. object member functions. First we have the *preamble*, which checks the actual input parameter values. The *execute* method, as its name suggests, contains the algorithm performing the useful stuff. Finally, the *postamble* checks the output parameter values. The preamble, execute and postamble are empty by default (no input or

output parameters) and the developer usually writes only the execute method to perform a given algorithm.

Note

Once parameters (input or output) receive a value, *they are automatically reset to their default values after the Task has been executed*. Note in particular that also *output* parameters are reset, so to keep a Task output for further inspection it has to be assigned to a variable upon execution, like this:

```
result = myTask()
```

One more thing to note is the possibility to define new default values for Task parameters. If we have a `myInput` integer parameter for our `myTask` Task, we can set its new default value to 42 like this:

```
myTask.setAsDefault("myInput", 42)
```

Now equipped with this knowledge we can turn our average algorithm into a Task.

4.2.3. An Example of a Task: Average

To turn our average algorithm into a Task we need to wrap the algorithm into a suitable piece of code.

We will name the task itself `Average` (a Task is a class, it is callable from the command line, and generally class names are capitalised nouns). In our `Average` class we have no needs other than setting up a signature and calling the average function as part of its execution.

One change from our function to our class is that we will explicitly have two parameters in the class definition. One (in a similar way as the function) is our input table, but for the class we declare a second parameter to hold the result of computing the average. As a requirement, we would like to change our original average function as little as possible.

In the next paragraphs we explain (with code and comments) what packages are necessary to import, how to define the Task (creation code), the method to perform a function (execute) and how we use and test the Task (with different parameter access methods).

4.2.3.1. Importing definitions

For our given code we need to import definitions that are used by our task:

```
# Import task framework classes.
from herschel.ia.task.all import *  # ❶
```

Some explanation about the import:

❶   Here we import all the task framework classes we need. Task and TaskParameter classes will be automatically imported with the `all` import statement.

Note that the preferred way to import the needed classes from the task framework is the so called 'all' import statement:

```
from herschel.ia.task.all import *
```

4.2.3.2. Creation

First the code for the creation method called `__init__` in python:

```
class Average(JTask):  # ❶
# Creation method
 def __init__(self,name = -"averageTable"):  # ❷
  p = TaskParameter("table",valueType = TableDataset, mandatory = 1)  # ❸
```

```
  self.addTaskParameter(p)  # ❹
  p = TaskParameter("result",valueType = Double1d, type = OUT)  # ❺
  self.addTaskParameter(p)
```

And some explanations about the code...

❶   Here we define a class `Average` which has `JTask` as a parent class. In other words, `Average` *inherits* from `JTask`. Note that `Jtask` is a python file and has no JavaDoc therefore.

❷   This line declares the creation method used by any instance of the `Average` class. `self` as the first argument represents the instance that we are currently working on. The `name` argument is the default value indicated (which the user can of course overwrite).

The rest of the code is the definition of the signature for the task `Average` and is as follows:

❸   This line creates a parameter whose name is `table`, data type is `TableDataset`. This is a mandatory parameter, i.e. an input parameter which must have a value before the algorithm is performed. The preamble will verify that the user has set a value for this parameter and will eventually warn the user that the execution of the task cannot take place.

❹   Here we add the parameter to the signature of this task.

❺   We proceed in a similar way for our second parameter (as mentioned above) which will hold the result of our computation. The only difference for the second parameter is the `type = OUT` statement which means that this parameter will hold an output value. As a side note the mode of parameters can be `IN`, `OUT` or `IO` (both input and output), the default being `IN`.

4.2.3.3. Execution

First we examine the code for the execution method called `execute` as predefined in the `JTask` base class. This simply follows on from the previous set of code that initiated the task and should be added to the end of it:

```
# Execute method itself
  def execute(self):  # ❶
   self.result = average(self.table)  # ❷
```

❶   This is a declaration stating that we define the method `execute`. Actually we redefine the empty `execute` method of `JTask`. This method has a parameter `self` which refers to the task we are currently working with, rather than to any other parts of the current IA session.

❷   This line means 'take this instance table value, perform the average operation on it and deliver the result to this instance result'. So in one line we perform the whole operation using our own actual parameters.

Together with the signature defined in the previous section we have set up our Task. The complete script should look like the Task Average (below). We now load this into our session.

```
# File: Average.py
#=========================================
# Import task framework classes.
from herschel.ia.task.all import *
from herschel.ia.task.JTask import JTask

class Average(JTask):
  #Creation method
  def __init__(self,name = -"averageTable"):
    #
    p = TaskParameter("table",valueType = TableDataset, mandatory = 1)
    self.addTaskParameter(p)
    p = TaskParameter("result",valueType = Double1d, type = OUT)
    self.addTaskParameter(p)
  # Execute method itself does the running of -'average'
  def execute(self):
    self.result = average(self.table)
```

Example 4.2. The Average Task

4.2.3.4. Usage

Below is the command line code to input into the HIPE Console view for testing our `Average` task. First we *instantiate* the `Average` class creating an object called `avg`:

```
avg = Average()
```

We are using the default name of `averageTable` for our Task. To change the name we would have written for instance `avg = Average("Simple average of table data set")` or `avg = Average(name = "Mine")`.

We can now formulate a table using the `createTable` routine in the set of three functions we created at the outset.

```
table = createTable()
```

The interesting part comes when we use the following:

```
print avg(table)
```

We have executed the Task and printed its result. To make sure that it indeed executed successfully, we can look at the `statusMessage`:

```
print avg.statusMessage
```

A more direct way to execute our Task would be

```
print avg(createTable())
```

On the other hand, we could do everything in a long-hand fashion, doing one little step at a time:

```
avg.table = table
avg()
result = avg.result
print result
```

Here we tell our average task that its input is called 'table'. The second line runs the task itself and we assign the result from this to a variable called 'result' in the third line. Finally, this result is printed.

4.2.3.5. Getting help on Tasks

If you stumble upon a task you have never used before you will probably want some way of finding out about is parameters, whether they are mandatory or not, and so on. Taking our Average task as example, if you type

```
info('Average')  # Note it's -'Average' with single quotes
```

you will be greeted by the following window:

Figure 4.1. Getting help on a Task.

It may appear fairly intimidating, but it provides a lot of useful information to users once they get past the initial shock. In particular, look at the sections called `Inputs:` and `Outputs:`. They list the input and output parameters, which are most of what is needed in order to use a Task. In particular, here we see that we have one input parameter called `table`, that it's a `TableDataset` and is mandatory (`Optional: false`). Similarly, we see that the Task will output a single `Double1d`. The information about `status`, `statusMessage`, `progress` and `views`, found in the lower part of the help window (not shown in the picture) is of limited interest to users.

What appears in the help window also depends on what developers originally put into the Task. For example, in our case we have the hardly reassuring `Task: null` and `Name: null` messages at the very top of the window. But if we give a name to our Task like this

```
avg.setName("My first Task")
```

we will see that after a short while the new information will appear in the help window.

4.2.3.6. Adaptations in the Preamble to a Script

The adaptation to the input of our Average script can be made in a preamble to the task, such as in the following script. Note that here we import the `task` classes one by one, just to show in detail what is needed.

```
# Importing JTask classes
from herschel.ia.task.all import *
# Other needed imports
from org.python.core import PyList
# And here is our AdaptAverage class
class AdaptAverage(JTask):
    # Creation method
    def __init__(self,name = -"Running Average"):
        p = TaskParameter("vector1",valueType = PyList, mandatory = 1)
        self.addTaskParameter(p)
        p = TaskParameter("vector2",valueType = PyList, mandatory = 1)
        self.addTaskParameter(p)
        p = TaskParameter("result",valueType = Double1d,type \
            = OUT)
        self.addTaskParameter(p)
        # Create an internal JTask variable -'table' which is our table data set
        self.__dict__['table'] = TableDataset()
    # In the preamble we do the adaptation from 2 vectors to one table
    def preamble(self):
        JTask.preamble(self)
        self.table["0"] = Column(Double1d(self.vector1))
        self.table["1"] = Column(Double1d(self.vector2))
    # Execute method itself
    def execute(self):
        self.result = average(self.table)
```

Example 4.3. The Adapt Average Task

In this example, the `from org.python.core import PyList` statement allows us to work with Python array lists (vectors). The task now takes two Python arrays and produces a table from the arrays with each array forming a column of the table. We then can run our `average` script on the table created in the preamble.

An internal instance variable is declared in the creation method with the statement: `self.__dict__['table'] = TableDataset()`.

Rewriting the preamble method. One should note that we first invoke the preamble from our parent task (JTask) to guarantee that our needed parameters do have a suitable value before putting them into the table.

The following short script can be used to test this adapted version of our averaging routine.

```
def test():
    sample1 = [1.0, 2.0, 3.0, 4.0]
    sample2 = [3.0, 4.0, 5.0, 6.0]
    avg = AdaptAverage()
    # Invocation using positional parameter
    print -'Result:', avg(sample1,sample2)
```

Input of the following command

```
test()
```

provides the following printed result

```
Result: [2.0,3.0,4.0,5.0]
```

4.2.3.7. Positional and Keyword Arguments in Tasks

Note

It should be noted that *positional* or *keyword* arguments can be used with tasks but NOT a mix of the two.

For example, the last line of our 'test' script effectively runs the following (try replacing the last line of the test() routine):

```
# Positional arguments
```

```
print -'Result:', AdaptAverage()(sample1, sample2)
# Keyword arguments
print -'Result:', AdaptAverage()(vector1=sample1, vector2=sample2)
# Since -'vector1' and -'vector2' are the two arguments for the
# AdaptAverage task.
```

Mixing of the two modes is ONLY allowed following all positional arguments. For example:

```
print -'Result:', AdaptAverage()(sample1, vector2=sample2)
```

But once keyword arguments start to be used then they must continue to be used. For example the
following code snippet will result in a compiling error when added to the 'test' program and recompiled.

```
print -'Result:', AdaptAverage()(vector1 = sample1, sample2)
# If this is added to -'test' and -"test' is then recompiled we get the
# following syntax error.
# SyntaxError: ('non-keyword argument following keyword',
# ('<string>', 6, 49, -''))
```

A similar syntax error occurs if the AdaptAverage() task was run on a single line outside of the 'test'
routine.

4.2.3.8. The `Transformer` example

Yet another JTask example. This one takes an array and transforms it into the first column of a
TableDataset. As before, the code comes with a testTran() function to check what the Task does.

```
from herschel.ia.task.all import *
from org.python.core import PyList

class Transformer(JTask):
    # Creation method
    def __init__(self, name = -'Vector Transformer'):
        p = TaskParameter(name = -"input", valueType = array(Integer), mandatory =
1)
        self.addTaskParameter(p)
        p = TaskParameter( name = -"result", valueType = TableDataset)
        p.type = OUT
        self.addTaskParameter(p)
    # Execute method
    def execute(self):
        self.result = TableDataset(description = -'Integrated vector as column
zero')
        r = Double1d(len(self.input))
        index = 0
        for data in (self.input):
            r[index] = data
            index = index + 1
        self.result['0'] = Column(r)

def testTran():
    sample = [10, 20, 30, 40]
    # Turn it into a table data set
    transform = Transformer()
    table = transform(sample)
    print -"Printing the table"
    print table
    print -"Printing the first column of the table"
    print table['0']
    print -"Printing just the data in the first column"
    print table['0'].data
```

Example 4.4. The Transformer Task

4.3. Tasks with graphical interfaces

This section describes how to handle GUI's and/or a dialog related to a task, how to check whether a
certain task supports the use of a dialog and/or GUI, as well as describing how to apply them.*It should*

be emphasised that the developer of a task needs to implement a dialog or GUI in the task. This section simply provides guidance to the user for using tasks that have dialog or GUIs included within them.

4.3.1. The use of task parameters handled via a dialog

In the case where a task includes a long or complex set of parameters a dedicated dialog can be provided by the original developer of the task. Such a component is handled by a boolean parameter called "dialog" which the user can invoke using

```
result = Task()(dialog=1)
```

Such a call results in a pop-up window which can be completed by selecting for example the "accept" button, which will close the GUI.

Note that all tasks in the future will include a boolean-parameter called "dialog". In cases where all the available input parameters are of the type String or Number (i.e. those the framework can handle for setting up a dialog) a dialog-popup will be provided, otherwise an exception is thrown.

4.3.2. The use of more enhanced GUIs

In case you have a more complex task or you want to re-execute a task several times using different inputs, a GUI might be introduced. Such a component is handled by a boolean parameter called "gui":

```
task = MyTask()
task.gui = 1 # gui interaction might  include an task.execute()
result = task.result # another gui interaction
result2 = task.result
```

Such a command sequence is very useful as it increases transparency. For example, the GUI might show the state of the parameters by including a field for each parameter and a plot or image representing the quality of the resulting output.

To summarise: the user of a task applies its views by the use of related the booleans (task parameters). In case of a one-time user interaction such a boolean is called "dialog" and otherwise it is called "gui". Note that in case more GUI components are involved additional booleans could be introduced, the task specific documentation should include this info.

4.3.3. Conventions for parameter labels

The names of task parameters in the task graphical interface are formatted according to these rules:

- The primary input parameter is first in the list and in **bold**.

- All the mandatory parameters, including the primary input, have an asterisk following their name (for example width*). Note that this does not change the actual parameter name.

- All other parameter names are written in plain text.

Chapter 5. Overview of DP packages

5.1. Introduction

To access functionality within HCSS packages you have to *import* it into your HIPE session. For many packages this is done automatically by default; if not you can do it manually via commands like the following:

```
from herschel.ia.numeric import *
```

There are several packages available within the HCSS. In this chapter we provide an overview of the main DP packages only. A full listing of packages and classes available in your HCSS installation is given in the API documentation, which you can access by selecting *HCSS Developer's Reference Manual (API)* from the HIPE Help System table of contents.

A number of DP packages are discussed elsewhere in some detail. The *Numeric* package was discussed in Chapter 3, while the *Plot* and *Display* packages are discussed in the *Data Analysis Guide*. Illustrations of how to use parts of several other HCSS packages are also shown in other chapters.

5.2. Overview of Javadoc Documentation for DP Packages

The javadoc is normally started up as three frames in a web browser as illustrated in Figure 5.1 The upper left frame contains the *packages index* which is a clickable list of all packages in the system. The title in that frame represents the HCSS build number for which this documentation is valid. The lower left frame contains the *classes index* which is a clickable list of all classes. The selection of classes shown in this frame depends on the package that was selected in the packages index frame. The *Main frame* contains overview information on the system and packages or shows the javadoc for a selected class.
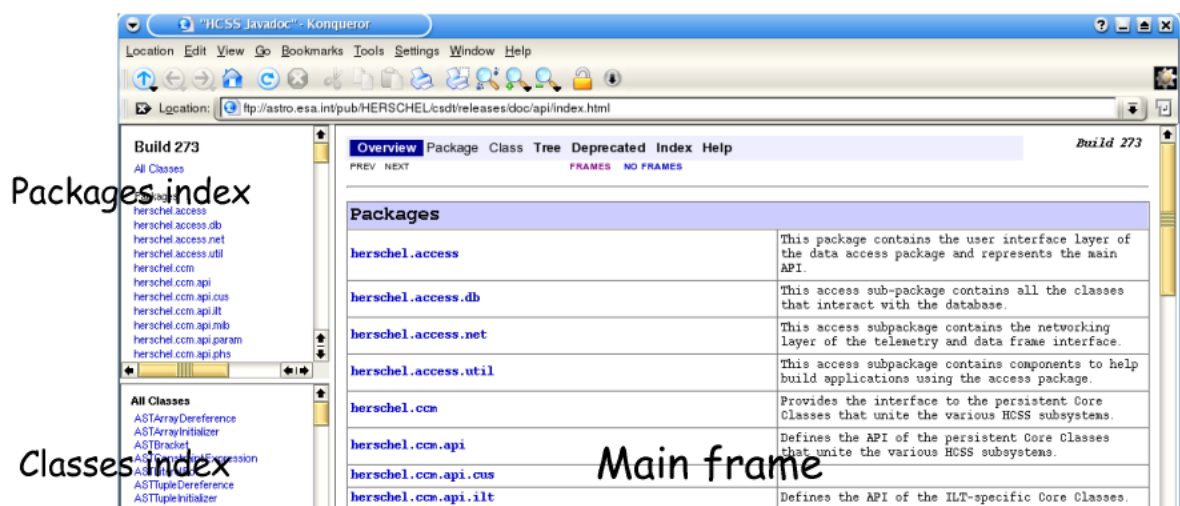
Figure 5.1. Web browser page of JavaDocs top level frame.

Click in the *Packages index* frame to select a package and update the *Classes index* frame to show those classes for the selected package. Click the *Classes index* frame to show the javadoc of a particular class in the *Main frame*.

The *Main frame* contains a kind of navigation bar at the top where the view in this frame can be selected. The figure above shows the overview of all the packages. Other views are: Package, Class, Tree, Deprecated, Index, and Help. These views will be explained in more detail below. In the overview the Package and Class views are disabled, they become available when a package or class is selected. Figure 5.2 shows the slightly expanded navigation bar for the Class view.

Overview Package **Class** Tree Deprecated Index Help
PREV CLASS   NEXT CLASS   FRAMES   NO FRAMES
SUMMARY: NESTED | FIELD | CONSTR | METHOD   DETAIL: FIELD | CONSTR | METHOD

Figure 5.2. Navigation bar on the *class view* of JavaDocs.

Note that the navigation bar provides the possibility to browse through packages and classes with NEXT and PREVIOUS and provides direct access to the specific parts of the class documentation e.g. constructors (start class/program) or methods (which can be thought of as sub-routine components of programs that can be applied). It is also possible to switch between FRAMES and NO FRAMES. With NO FRAMES only the *Main frame* of the javadoc will be shown and index frames become unavailable.

5.3. Package view

Each package has a page that contains a list of its classes and interfaces, with a summary for each. This page can contain four categories*: Interfaces summary, Classes summary, Exceptions and Error summary*. Not all categories are always present. At the end there is the package description and possible links to specific and/or related documentation.

Figure 5.3 shows the `herschel.ia.dataset` package which contains a number of interface and classes e.g. *Dataset* and TableDataset. You can see that the *Classes index* frame provides a clear separation of interfaces and classes and the *Main frame* shows the interface and class summaries and provides a brief package description with links to package specific info at the bottom (The image of the *Main frame* has been manipulated to shows the categories available without too much cluttering the picture). You can navigate to the interface and class detailed documentation by clicking the names in the summary tables or in the *Classes index* frame.

Figure 5.3. Package description page in JavaDocs.

5.4. Class view

Each class and interface has its own separate page in the *Main frame*. Each of these pages has three sections consisting of a class/interface description, summary tables for constructors and methods, and detailed descriptions of constructors, methods and attributes. The information shown in the class view is restricted to the public *API (Application Programming Interface)*.

Each summary entry contains the first sentence from the detailed description for that item. The summary entries are alphabetical, while the detailed descriptions are in the order they appear in the source code. This preserves the logical groupings established by the programmer.

Figure 5.4 is taken from the *Main frame* of the *TableDataset* class and shows the class description together with its hierarchy. You can see that the *TableDataset* implements a number of interfaces and also has one known sub-class i.e. *SpectrumDataset*. The second part of the figure shows a more detailed description of the class usage. This description is provided by the programmer in the source code.

```
herschel.ia.dataset
Class TableDataset


java.lang.Object
  └herschel.ia.dataset.AbstractAnnotatable
      └herschel.ia.dataset.TableDataset

All Implemented Interfaces:
      Annotatable, Attributable, Dataset
Direct Known Subclasses:
      SpectrumDataset
_____

public class TableDataset
extends herschel.ia.dataset.AbstractAnnotatable
implements Dataset

A TableDataset is a tabaular collection of Columns. It is optimized to work on array Data
as specified in the herschel.ia.numeric package.

This approach is convenient in many cases. For example, one has an event list, and each
algorithm is adding a new field to the events (i.e. a new column.

The orthogonal approach (adding rows) is therefore expensive and therfore currently no
mechanism is implemented to add rows to the table.

Jython usage:
creation:

    creation:
    $ x=TableDataset(description="This is my table")
    $ x["Time"]=Column(data=time, quantity=SECONDS)
    $ x["Energy"]=Column(data=energy, quantity=ELECTRON_VOLTS)
```

Figure 5.4. The class view of TableDataset showing a brief description and a short example of its usage.

Scrolling down in the *Main frame* brings you to the summary section which is shown in Figure 5.5. The constructor summary shows all public constructors for this class with their specific argument list. To see detailed information on the constructor click the name of the constructor that you need. Constructors are methods that create objects of a particular type. The code example in the description section above shows you how to create a *TableDataset* on the *jython* command line.

Figure 5.5. Page showing the constructor mechanism (how to create a TableDataset) and the associated set of methods (what you can do with the TableDataset you created).

The method summary shows all public methods for this class in alphabetical order. For detailed information on a specific method, click its name. In this method summary there are a number of things to note. The return values of the methods are in the left column while the method signature and a summary line is in the right column. The summary line can be preceded with a **deprecation** note. Deprecation means that this method should not be used anymore because it is marked to be removed from future releases. The deprecation comment normally provides the alternate or new method to be used instead. An overview of all deprecated methods in the whole system is available from the navigation bar at the top of the *Main frame*.

Sometimes method names can start and end with two underscore characters like in `__getitem__` above. These methods are special constructs which allow you to use the specific jython syntax to access and manipulate objects from this class.

5.5. Other views

5.5.1. Tree view

There is a Class Hierarchy page for all packages, plus a hierarchy for each package. Each hierarchy page contains a list of classes and a list of interfaces. The classes are organised by inheritance structure starting with *java.lang.Object*. The interfaces do not inherit from *java.lang.Object*. When viewing the Overview page, clicking on "Tree" displays the hierarchy for all packages. When viewing a particular package, class or interface page, clicking "Tree" displays the hierarchy for only that package.

5.5.2. Deprecated view

The Deprecated API page lists everything that has been *deprecated*. A deprecated API is not recommended for use, and a replacement API is usually suggested.

> **Warning**
>
> Deprecated APIs may be removed in future versions.

5.5.3. Index view

The Index contains an alphabetic list of all classes, interfaces, constructors, methods, and fields.

5.6. DP Packages And Documentation

The following short paragraphs outline the packages currently available within the Herschel DP system. For full details please see the Javadoc.

5.6.1. herschel.ia.dataflow

Handles processing threads. Particularly useful for Quick Look Analysis (QLA) and Standard Product Generation (SPG). It can be used in interactive sessions too. Allows the user to connect scripts from process modules as is typically required for a set of data reduction steps. Dataflow also supports event-based processing as well as threads.

Main subpackages:

- **herschel.ia.dataflow.data.process:** Classes for handling the processes used in a dataflow session.

- **herschel.ia.dataflow.example.indicator_control.monothread:** Classes used to illustrate the control of a dataflow.

- **herschel.ia.dataflow.example.indicator_control.multithread:** Same as above, but for multiple threads.

- **herschel.ia.dataflow.template:** Class to allow template dataflow to be created.

- **herschel.ia.dataflow.util:** Class for identifying dataflows.

5.6.2. herschel.ia.dataset

Contains *Table Datasets*, *Array Datasets*, *Composite Datasets*, *Products* and all auxiliary components such as columns, parameters and metadata. Datasets and products are described in Chapter 2.

Main subpackages:

- **herschel.ia.dataset.demo:** Contains classes that demonstrate the use of datasets.

- **herschel.ia.dataset.gui:** Contains the *Dataset Inspectro* graphical interface.

- **herschel.ia.dataset.image:** Provides a framework for defining images, cubes of images and stacks of images. Includes tools for adding World Coordinate System information.

- **herschel.ia.dataset.history:** Defines the *History Dataset*, which records the complete history of the tasks which were executed to produce a Product.

- **herschel.ia.dataset.spectrum:** Contains tools for defining one- and two-dimensional spectra, and spectral cubes.

5.6.3. herschel.ia.demo

Contains demonstration scripts.

> **Warning**
>
> Many of these scripts may be out of date and not work with recent versions of HIPE.

5.6.4. herschel.ia.doc

Contains developer-oriented documentation in HTML format. Contents of this package are also available from within the HIPE Help System.

> **Warning**
>
> The Javadoc available in this package is incomplete. Please access the Javadoc from the HIPE Help System

5.6.5. herschel.ia.document

Provides tools to generate documentation of dynamic as well as static DocBook documents in different formats.

5.6.6. herschel.ia.gui

Contains several subpackages related to graphical applications.

Main subpackages:

- **herschel.ia.gui.apps:** Contains the classes used to build graphical applications such as HIPE.

- **herschel.ia.gui.cube:** Graphical interfaces to analyse data cubes.

- **herschel.ia.gui.explorer:** Graphical interfaces to analyse datasets, such as TablePlotter and OverPlotter.

- **herschel.ia.gui.image:** Classes for handling images. The display capabilities from this package are discussed in the *Data Analysis Guide*.

- **herschel.ia.gui.plot:** Plotting utilities. For more details see the *Data Analysis Guide*.

5.6.7. herschel.ia.inspector

Contains the classes and utilities for providing the dataset and session inspectors available in HIPE (see Section B.3.5).

5.6.8. herschel.ia.io

Provides a means of accessing local archives where Products can be saved or loaded from. Products are combinations of data and information and can be likened to the contents of a single FITS file.

Main subpackages:

- **herschel.ia.io.ascii:** Allows input and output of data to and from ASCII files.

- **herschel.ia.io.fits:** A FITS implementation that can write Products to a FITS file and read such FITS files back into the system. Allows the production of a FITS archive.

- **herschel.ia.io.dbase:** Allows data/products to be put into objects that can be stored in databases (Versant databases are currently available for use with the HCSS).

5.6.9. herschel.ia.jconsole

Contains the classes used in running JIDE, a legacy application for running and editing of Jython scripts, developed before HIPE. Allows control of the JIDE setup and access to classes that setup the components of the GUI interface (in *herschel.ia.jconsole.gui*).

5.6.10. herschel.ia.numeric

Contains numeric and mathematical tools described in Chapter 2 and Chapter 3

Main subpackages:

- **herschel.ia.numeric.toolbox:** Provides a large set of numeric classes. These include mathematical functions (trigonometric functions, polynomials), Fourier transforms, fitter functions, interpolation and matrix functions. Note that these classes are automatically loaded when starting HIPE.

  This package contains the following subpackages:

  - **herschel.ia.numeric.toolbox.basic:** Provides classes that allow basic mathematical manipulation of numeric arrays: trigonometric functions, mathematical product, variance and so on.

  - **herschel.ia.numeric.toolbox.filter:** Provides the classes `BoxCarFilter`, `Convolution` and `GaussianFilter`.

  - **herschel.ia.numeric.toolbox.fit:** Provides classes that allow the fitting of data with numerous models (iterative fitters, sine model fitters, polynomial model fitters and so on).

  - **herschel.ia.numeric.toolbox.integr:** Provides integrator functions for several integral models (Gauss-Jacobi, Gauss-Laguerre and so on).

  - **herschel.ia.numeric.toolbox.interp:** Provides classes that allow the interpolation of data. These include `Interpolator` (a general interpolator), `LinearInterpolator`, `CubicSplineInterpolator` and `NearestNeighborInterpolator`.

  - **herschel.ia.numeric.toolbox.mask:** Provides tools for creating and managing masks, in particular the two classes `FixedMask` and `PackedMask`.

  - **herschel.ia.numeric.toolbox.matrix:** Provides classes that allow the manipulation of `Double2d` arrays holding matrices. It includes the classes `MatrixDeterminant`, `MatrixInverse` and `MatrixSolve`.

  - **herschel.ia.numeric.toolbox.random:** Provides tools for generating pseudo-random numbers with uniform (`RandomUniform`), Gaussian (`RandomGauss`) and Poisson (`RandomPoisson`) distributions.

- **herschel.ia.numeric.toolbox.util:** Provides the classes `MoreMath`, which has methods for mathematical manipulation of single numerical elements (integers, doubles, bytes and so on), and `Util`, which has utilities for converting arrays.

- **herschel.ia.numeric.toolbox.xform:** Provides the classes `FFT`, `Hamming` and `Hanning` for Fourier transforms and Hanning/Hamming smoothing of data.

5.6.11. herschel.ia.obs

Defines the *Observation Context*, a container for Products applicable to a specific obervation, and related classes.

Main subpackages:

- **herschel.ia.obs.auxiliary:** Defines the auxiliary Products related to an observation, and their container, the *Auxiliary Context*.

- **herschel.ia.obs.cal:** Calibration-related classes.

- **herschel.ia.obs.quality:** Defines the *Quality Context* and the flags used for quality control.

5.6.12. herschel.ia.pal

Defines the *Product Access Layer*, which allows storage and retrieval of Products both locally and remotely. The Product Access Layer is treated in detail in [Appendix A](#).

Main subpackages:

- **herschel.ia.pal.browser:** Defines the *Product Browser* graphical application.

- **herschel.ia.pal.io:** Defines classes for importing and exporting Products to FITS format.

- **herschel.ia.pal.pool:** Defines, in various subpackages, the available types of *Product Pools*.

- **herschel.ia.pal.query:** Defines the types of query that can be applied to a *Product Storage*.

5.6.13. herschel.ia.pg

Describes the *Product Generation Framework*, on which running of instrument pipelines is based.

Main subpackages:

- **herschel.ia.pg.od:** Defines the *Operational Day Plugin*, used to process an entire OD *before* processing its observations.

- **herschel.ia.pg.plugins:** Defines basic versions of other plugins that are applied during pipeline processing, such as `BasicLevel0Plugin` and `BasicQualityPlugin`.

5.6.14. herschel.ia.qcp

Defines components and utilities to handle Quality Control messages.

Main subpackages:

- **herschel.ia.qcp.example:** Provides an example Task for using the facilities of this package.

- **herschel.ia.qcp.flags:** Provides a hierarchical structure of Quality Control flags.

- **herschel.ia.qcp.gui:** Provides graphical components for displaying Quality Control messages.

- **herschel.ia.qcp.plugin:** Provides plugins for logging Quality Control messages during Operational Day and pipeline processing.

- **herschel.ia.qcp.tools:** Provides a standalone application for displaying Quality Control information.

5.6.15. herschel.ia.spg

Manages the execution of the data reduction process for all the instrument in the Herschel satellite. It is built upon the framework defined in the herschel.ia.pg package (see Section 5.6.13).

Main subpackages:

- **herschel.ia.spg.gui:** Contains the *Pipeline Manager* graphical interface.

- **herschel.ia.spg.kayako:** Contains a helper class for creating a ticket in the *kayako* system.

- **herschel.ia.spg.od:** Tools for scheduling and executing Operational Day processing.

- **herschel.ia.spg.ops:** Miscellaneous tools for configuring pipeline processing.

- **herschel.ia.spg.tools:** Classes for memory monitoring and the remote management of processing queues.

5.6.16. herschel.ia.task

herschel.ia.task Provides the tools needed to create a data processing *Task* which you can then incorporate into your scripts. Tasks have an associated *signature* (parameter setup); in setting up a Task, parameter checks can be performed and a history of the processing can be kept.

This package is discussed in Chapter 4.

Main subpackages:

- **herschel.ia.task.example:** Provides example Tasks that demonstrate some features of the package.

- **herschel.ia.task.gui:** Provides components used to build graphical interfaces for Tasks.

- **herschel.ia.task.history:** Provides a class for managing the history of a Task.

- **herschel.ia.task.mode:** Provides different execution modes for a Task (interactive, on demand, systematic and test).

- **herschel.ia.task.util:** Miscellaneous utility functions for Task development.

5.6.17. herschel.ia.toolbox

Provides tools for a wide range of data analysis needs. Tools are organized in thematic subpackages.

Main subpackages:

- **herschel.ia.toolbox.astro:** Astronomical utilities.

- **herschel.ia.toolbox.cube:** Tasks for importing and analysing data cubes.

- **herschel.ia.toolbox.fit:** Tasks for function fitting.

- **herschel.ia.toolbox.hsa:** Provides an interface for accessing the Herschel Science Archive.

- **herschel.ia.toolbox.image:** Tasks for image processing (cropping, smoothing and so on).

- **herschel.ia.toolbox.mapper:** Tasks for mapmaking.

- **herschel.ia.toolbox.pointing:** Provides a task for plotting pointing information.

- **herschel.ia.toolbox.spectrum:** Tasks for analysing spectra. This package contains several subpackages, among which are the following:

  - **herschel.ia.toolbox.spectrum.fit:** Tools for fitting spectra.

  - **herschel.ia.toolbox.spectrum.gui:** Tools for visualising spectra.

  - **herschel.ia.toolbox.spectrum.operations:** Tools for performing mathematical operations on spectra (divide, average, resample and so on).

  - **herschel.ia.toolbox.spectrum.projection:** Tools for projecting spectral data on the sky.

  - **herschel.ia.toolbox.spectrum.selections:** Tools for selecting and managing ranges and discrete values within spectra.

  - **herschel.ia.toolbox.spectrum.standingwaves:** Tools for fitting and removing fringes.

  - **herschel.ia.toolbox.spectrum.utils:** Other utilities, for example to integrate and interpolate spectra.

- **herschel.ia.toolbox.srcext:** Tools for point source extraction.

- **herschel.ia.toolbox.trend:** Tools to support trend analysis processing. See this TWiki page for more details.

- **herschel.ia.toolbox.util:** Miscellaneous tools, among which are tasks for importing from and exporting to ASCII tables and FITS files.

5.6.18. herschel.ia.vo

Contains tools that implement the interface to the Virtual Observatory.

Chapter 6. Time measurement

6.1. Introduction

This note describes which and how time is defined within HCSS and how to deal with it. Unfortunately, there are several ways in which time can be represented. The standard for the HCSS/DP is a `FineTime` - which is the number of microseconds since the beginning of 1 January 1958. This provides the kind of accuracy needed to represent time on a space mission.

However, there are several other time representations and it is often the case that conversions between times/dates is necessary. In particular, it is noted that the standard Java commands lead to date measurements with respect to 1 January 1970. This chapter indicates how to deal with times within DP and converting between the various times, particularly between dates and `FineTime`'s.

6.2. Time Definitions

6.2.1. System time in DP

There are many ways to access the system time within DP. See also the description of the Java class "Date" for a discussion of slight discrepancies that may arise between "computer time" and coordinated universal time (UTC).

The Java `Date` class is deprecated and is being replaced by a more flexible `SimpleDateFormat` capability within Java that allows the user to express dates more conveniently. A `Date` object is still obtained and can be turned into a `FineTime` (see below) once created.

Two possibilities for creating a "Date" object are:

```
# To get the current time in milliseconds:
# The difference, measured in milliseconds, between the current
# time and midnight, January 1, 1970 UTC.
print java.lang.System.currentTimeMillis()
# To get the number of milliseconds since
# January 1, 1970, 00:00:00 GMT represented by a Date object.
d = java.util.Date()
#printing this gives the current time and date at the location of the
#system on which the java is being run.
print d
#We can also get the number of milliseconds since Jan 1, 1970 using
#this Java Date
print d.getTime()
```

Example 6.1. Current Time

Note that while the unit of time of the return value is a millisecond, the granularity of the value depends on the underlying operating system and may be larger.

If we want to get the number of milliseconds since 1 January 1970 for any other date then we can use a non-default form of the Java Date capability where the year, month, day, hour, minute and second are provided.

- Year format -- year (A.D.) - 1900. So the year 2006 = 2006 - 1900 = 106

- Month format -- number of the month, beginning from January = 0. E.g. March = 2.

- Day -- just day number in the month.

- Hours, minutes, seconds -- on the 24-hour clock.

NOTE: This is the time on our computer system.

```
#Format of date is year (in units of true year -- 1900), month (number 0...11),
#day, hour, minute, second. So the following gives us the number of milliseconds
#between the beginning of 1 January 1970 and 3:15:00 pm on 23 October 2004.
d = java.util.Date(104, 9, 23, 15, 15, 0)
print d # should indeed show we have 3:15pm on 23 October 2004
print d.getTime() # provides the number of milliseconds between this
#date and 1 Jan 1970.
```

The following sample code shows how to use `SimpleDateFormat` to create a "Date" object.

```
simpleDate = java.text.SimpleDateFormat("yyyy.MM.dd HH:mm:ss z")
#set up how you want to set up your input Date format. In this
#case we could input -"2006.01.14 01:00:00 CST" for 1a.m. on 14
#January 2006. z --- indicates the time zone (default is the zone for the
#computer system being used).

simpleDate.applyPattern("dd/MM/yy HH:mm")
#change the pattern to a different format

startTime = simpleDate.parse("13/01/06 14:06")
#create the data object -"startTime"

print startTime
#...and see what this looks like
```

Allowed choices for the data format are available from Java documentation of the SimpleDateFormat capability.

6.2.2. International Atomic Time (TAI) and `FineTime`

TAI is an international standard measurement of time based on the comparison of many atomic clocks. TAI is the basis for Coordinated Universal Time (UTC). `Finetime` is based on TAI as measured from 00:00:00 1 January 1958.

6.2.3. Coordinated Universal Time (UTC)

UTC , World Time, is the standard time common to every place in the world. UTC is derived from International Atomic Time (TAI) by the addition of a whole number of "leap seconds" to synchronise it with Universal Time 1 (UT1), thus allowing for the eccentricity of the Earth's orbit and the rotational axis tilt (23.5 degrees), but still showing the Earth's irregular rotation, on which UT1 is based.

6.2.4. DecMec Time [PACS only]

The commands DPUSelectTime and DPUWriteTime are selecting and setting a start time which is written to the TMP1 and TMP2 fields of the Dec/Mec headers. This is used in coordinating the activities of the mechanical devices on board PACS. It is possible to construct an absolute time by adding counters (CRDC) to the start time considering an offset between setting and writing the start time.

This offset is expected to be a number with an uncertainty depending on the system load. It might require a calibration file. Currently this offset is not considered.

In case the commands and are not given the TMP1 and TMP2 fields are zero. To avoid software confusions the time will be related to a fixed date (1.Jan 1970, 0:00).

During construction of the SpuBuffer the time is computed from the TMP1, TMP2 entries in the Dec/ Mec header and the CRDC counter. This time is used during construction of the DataFrameSequence and the associated Tables holding the SPU science data.

Between the Dec/Mec time and the packet time (see PusTmBinStruct) we have an offset. Therefore the association between HK and science data will be within an accuracy of 2 seconds.

6.3. Time in Instrument House-Keeping (HK) Data

The most convenient method of obtaining time stamped HK information is the use of the "herschel.binstruct" package. The use of this is illustrated in Chapter 12.10 where HK data is obtained from a database and then read/converted for use within the DP environment.

When dealing with HK time information directly, it is important to know that telemetry packets contain the time as defined within the "PUS Data Field Header". The field represents the on-board reference time of the packet, referenced to TAI, expressed in spacecraft time units - CCSDS Unsegmented Time Code (CUC) units. CUC units are multiples of 1/65536 sec from 1 January 1958 in TAI time. CUC units cannot be expressed in whole microseconds but can be converted to the *FineTime* standard (see below).

CUC time is written for HK by the data processing unit (DPU).

Current `PusTmBinStruct` methods related to time:

long getTime()

Returns the packet time of the Pus telemetry packet.

boolean isTimeSynchronized()

Returns true if the telemetry packet is synchronized, false otherwise.

java.util.Date getTimeAsDate()

Returns the packet time as a Date object.

FineTime getTimeAsFineTime()

Returns the packet time of the Pus telemetry packet as FineTime.

6.4. Time conversion

6.4.1. Time conversion in HCSS

It can often be the case that we need to convert between FineTime (TAI) and Date (UTC). Coordinated Universal Time is expressed using a 24-hour clock and uses the Gregorian calendar. FineTime represents a TAI time (epoch 1958), whereas the Java Date class is used to represent UTC, by resetting the system clock whenever a leap second occurs and don't need to handle leap seconds. Converting between Java dates and the FineTime standard requires the use of the DateConverter() class. Long integers can also be directly converted to FineTimes and are interpreted as representing the number of microseconds since 00:00:00 1 January 1958. In Example 6.2 we illustrate how to create a FineTime from a long integer and convert back and forth between FineTime and Java Dates.

```
from herschel.ccm.util import *
from herschel.share.fltdyn.time import *

# FineTime to Date
# Enter a time in seconds (a long integer -- put letter -"l"
# at the end of the number)
c = FineTime(1436094449715400l) # convert to a FineTime
# Prints corresponding date and time
print DateConverter.fineTimeToDate(c)
# Date to a FineTime
d = java.util.Date() # gets today's date and time
# Prints corresponding FineTime
print DateConverter.dateToFineTime(d)
```

Example 6.2. Time conversion between `Date` and `FineTime`

6.4.2. CucConverter

Converts between Spacecraft Elapsed Time, in CCSDS Unsegmented Time Code (CUC) format and FineTime (TAI). This implementation is for the Herschel CUC format, which is corrected on-board the spacecraft to TAI (epoch 1 Jan 1958). This representation uses 32-bits for seconds and 16 bits for fractional seconds. CUC times are multiples of 1/65536 sec and cannot be expressed as an exact multiple of 1 microsecond (the resolution of FineTime). However, the following relations hold for 'coarse' and 'fine' values in the allowed range:

long coarse(FineTime t)

Return the number of whole seconds since the epoch 1 Jan 1958.

long cucValue(FineTime t)

Return the number of 1/65536 fractional seconds since the epoch 1 Jan 1958.

int fine(FineTime t)

Return the fractional part of the number of 1/65536 seconds since the epoch 1 Jan 1958.

FineTime toFineTime(long cuc)

Return a new FineTime constructed from a 48-bit CUC time.

FineTime toFineTime(long coarse, int fine)

Return a new FineTime constructed from CUC coarse & fine fields.

```
from herschel.share.fltdyn.time import *

d=CucConverter.toFineTime(50000000000000L)
#Converts the long integer -- representative of a CUC time --
#into a FineTime. The FineTime is stored in d.
e = CucConverter.coarse(d)
#provides the number of whole seconds since 1 Jan 1958
#and stores it in e.
print e
```

Appendix A. Advanced Product Access Layer

Warning

The internal format of local store indexes has changed in HIPE 3.0. To update your local stores to the new format, please read the instructions in the *Data Analysis Guide*: Section 1.2.2.1. **Failure to do so may result in permanent corruption of your data.**

The Product Access Layer (PAL) allows you to create and access *Product Pools*. Product Pools are data storage areas that could be on your laptop (a local store) or on a remote system. Examples of a remote pool are:

- The Herschel Archive

- Products accessed from a Versant database

- A pool which you can share with others on a remote computer

A useful component of the PAL is the *Product Browser*. This is a graphical visualisation tool covered in Section A.14. We will show an example of how to launch it from a HIPE session.

A.1. Product Storage

A *Product Storage* is the front-end interface that allows you to communicate with Products stored in pools.

Simply by *registering* a pool to your storage, you can access the Products in that pool.

A Product Storage provides mechanisms to load, save and query Products in the registered pools. When doing so you receive a reference to a Product (returned by the load() and save() commands) or a set of Product references (when querying). This functionality of a Product reference is provided by the `ProductRef` class; it allows to fetch information of the Product, such as metadata, without loading the Product in question in your memory completely.

A.1.1. Creating a storage and registering pools

You can create a storage as follows:

```
storage=ProductStorage()
```

Then you have to assign the Product pools that you want to access. You have to register at least one pool:

```
storage.register(SerialPoolClient("abc.xyz.org",123,"dummy"))
:
storage.register(poolN)
```

A.1.2. Saving and restoring Products

Saving a Product:

```
# Create a dummy product
product=Product(creator="Me")
product["array"]=ArrayDataset(data=Int1d.range(5))

# Saving the product returns a reference
```

```
reference=storage.save(product)
print reference.urn
# urn:simple.default:herschel.ia.dataset.Product:0
```

Loading a Product:

```
reference=storage.load("urn:simple.default:herschel.ia.dataset.Product:0")
```

A reference provides access to parts of the product as well as access to the product itself:

```
print reference.urn
# urn:simple.default:herschel.ia.dataset.Product:0

print reference.type
# herschel.ia.dataset.Product

meta=reference.meta
print meta["creator"]
# Me

product=reference.product
print product.creator
# Me
```

A.2. Product Pools

Before you can do something useful with a Product Storage, you have to register one or more pools to that store.

Product pools can load, save and query simple Products. All pools share the same features (the so-called `ProductPool` interface) such that they can be registered to a Product Storage.

Typically you set up one Product Storage and register one or more Product pools to it. However the design permits to create multiple Product Storages with a different registry of Product pools. Product pools can also be shared between two Product Storages.

Two main pools are available (`LocalStore` and `DbPool`), plus some mechanisms for setting up and accessing remote pools:

- A *LocalStore* for storing and accessing Products in your local system (default is FITS format).

- A *DbPool* for accessing Products from a remote object database, such as a Versant database.

- A *SerialClientPool* to read/write or access a remote pool. When used in conjunction with a *PoolDaemon* (which runs on the machine of the remote pool) this can make the remote pool immediately available to your session.

- A *CachedPool* is a way to cache everything retreived from a pool. It is useful if the pool you are working with is a remote on-line pool, and you want to work offline.

- A *HsaReadPool* to access the Herschel Science Archive (HSA).

- A *HttpClientPool*, a networked pool similar to `SerialClientPool`.

In the next few sections we will discuss and provide examples of pools mainly in the context of Local pools, but most of these examples can be generalized to any kind of pool. In later sections we will describe these other kinds of pools and some other useful concepts that refer to them.

A.3. Local Pools

We will in this subsection discuss Local pools. However much of this information presented here is applicable generally to any kind of pool.

A.3.1. The Default Local Pool directory and how to change it

By default, data is stored in a directory with the user-supplied store name in the following directory

```
home/.hcss/lstore/
```

This can be changed by changing the property `hcss.ia.pal.pool.lstore.dir`. There are two ways to do this:

1. Issue the following statement in the *Console* view of HIPE:

```
HIPE>
Configuration.setProperty("hcss.ia.pal.pool.lstore.dir", -"${user.home}/.hcss/
alternate_store/")
```

2. Add this line to the `hipe.props` file, located in the `.hcss` directory within your home directory:

```
hcss.ia.pal.pool.lstore.dir=${user.home}/.hcss/alternate_store/
```

If the `hipe.props` file does not exist, just create it.

If you use the first method, the property value will be reset to its original value the next time you start HIPE. If you use the second method, the property will be set permanently and will be available since HIPE start up.

Note

The local store directory can also be a link to another directory. This is useful if you want to store your products in a different hard disk with more space.

A.3.2. Registering Local Pools

The storage location pointed to by `hcss.ia.pal.pool.lstore.dir` can contain several pools, which in the specific implementation of local store are subdirectories in that location. After importing the PAL classes with from `herschel.ia.pal import *`, we create a storage object with `storage=ProductStorage()`. We obtain a reference (pool1) to a pool from the pool manager using the statement *pool reference* = `PoolManager.getPool(`*poolname*`)`, where *poolname* is a string. Then the pool reference is registered by `storage.register(`*pool reference*`)`. With the command `print PoolManager.getPoolMap()` we can see which pools are currently registered.

A practical example where we open two pools would look like this:

```
from herschel.ia.pal import *
storage = ProductStorage()
pool1 = PoolManager.getPool('default')
pool2 = PoolManager.getPool('test')
storage.register(pool1)
storage.register(pool2)
print PoolManager.getPoolMap()
```

In case there is already a pool with that name in the default directory, it is registered and becomes accessible. If it doesnt exist, the pool is created as soon as we store a product there. This can be verified by inspecting the respective directory before and after.

Tip

You can rename a local pool by renaming the corresponding directory, but *only if the pool was created with HCSS 3.0 or newer*.

At this point we have created a storage and opened two pools. Note that when writing to the storage, the data is written to the first pool that was registered. If you want to write to a different pool you can create and use another storage for writing, where you register the desired pool. The same pool can be registered with more than one storage at the same time. Here an example where we make the pool `"test"` accessible for saving products.

```
otherStorage = ProductStorage()
otherStorage.register(PoolManager.getPool('test'))
```

We should also note that storage can also be obtained with the `LocalStoreFactory`, however this is discouraged by developers who strongly recommend using the `PoolManager`.

A.3.3. Saving products in pools

Let us first create some products to play with. In this case we will create two products containing one table dataset each. First the table datasets are created from random numbers.

```
r = RandomGauss()
n = 1000
tbl1 = TableDataset(description='Test Dataset 1')
tbl1['time'] = Column(Double1d.range(n))
tbl1['signal'] = Column(Double1d(n).apply(r))
tbl1['error'] = Column(Double1d(n).apply(r) * 0.3)
prod1 = Product(creator='ThatsMe', description='Test Product 1')
prod1['Table1'] = tbl1
```

Well do the same for a second product:

```
tbl2 = TableDataset(description='Test Dataset 2')
tbl2['time'] = Column(Double1d.range(n))
tbl2['signal'] = Column(Double1d(n).apply(r))
tbl2['error'] = Column(Double1d(n).apply(r) * 0.5)
prod2 = Product(creator='ThatsMe', description='Test Product 2')
prod2['Table1'] = tbl2
```

Now we have two products, `prod1` and `prod2`, at our disposal. Their contents can be verified by launching the dataset inspector. Any product can be saved in our storage using the following statement: *urn* = Storage.save(*product*), where *product* is the product to be saved and *urn* is the resulting *Uniform Resource Name* that is a unique identifier of the product within the storage. This URN can be used directly to retrieve the product from the storage, however typically the URN is not remembered, but rather re-obtained by a query to the storage. This will be shown later.

Let us save our two products using:

```
urn1 = storage.save(prod1)
urn2 = storage.save(prod2)
```

To see how the URN looks just use:

```
print urn1, urn2
```

As they are written by default to the first registered pool of storage, they will end up in the pool named `default`. Let us store one of the products also in the pool named `test` using:

```
otherStorage.save(prod1)
```

As we will recover the URN of this product later by a query, we dont bother to store the URN right now.

A.3.4. Finding out what is in a storage

If you have followed all previous examples, there should be now three new products in your storage that have listed as creator *ThatsMe*. Two of the products should be in the first pool named *default*, while the third product should be found in *test*.

To find out the contents of a storage, you execute a *query* on it. The following example looks for products with *ThatsMe* as creator:

```
query1=Query("creator == -'ThatsMe'")
res = storage.select(query1)
print res
```

Now `res` contains a list of references to the products that satisfy the query. Printing `res` will give a list of URN values:

```
HIPE> print res
[urn:default:herschel.ia.dataset.Product:0,
urn:default:herschel.ia.dataset.Product:1, urn:test:herschel.ia.dataset.Product:0]
```

If you want to execute an unconditional query to find all products in our storage, you can use the following:

```
query2=Query("True")
res2 = storage.select(query2)
print res2
```

In case you have used the default storage before, there may be other products here that would now show up in the list.

For information on how to inspect a storage via the *PAL Browser Perspective* in HIPE, see the *Data Analysis Guide*: ????.

A.3.5. More on storage queries: Other kinds of query and more examples of command line queries

The Product storage can handle three types of queries:

- Attribute query is a (fast) query on meta data that **all** Products contain: *creator, creationDate, startDate, endDate, instrument, modelName*. This is akin to querying a standard set of FITS header keywords.

- Meta data query is a (semi fast) query on meta data that can be different from Product to Product, depending on what was placed in the product by the person creating it in the first place. This is akin to doing a query on any FITS keywords (if present).

- Full query is a data mining query that allows querying on *all* data elements in Products, using the general methods provided for Products and datasets as well as the additional methods provided in specialisations of those datasets and Products.

All query types have the same syntax, but a different purpose as described above. Setting up a query is as follows:

```
#Simple query
query = Query(expression)
#More advanced queries
query = AttribQuery(product-class, variable, expression)
query = MetaQuery(product-class, variable, expression)
query = FullQuery(product-class, variable, expression)
```

where the parameters to the query are:

- `product-class`: restricts a family of products. All Product classes have `herschel.ia.dataset.Product` as the base class. You can restrict the query to a sub-family of Product. For example, if all HIFI Calibration Product classes stem from `HifiCalProduct`, you can limit your search by specifying that class.

- `variable`: is a string denoting the variable name of the product that will be used in the expression.

- `expression`: is a string holding the query expression, which is limited to the query type.

It is worthwhile mentioning that the syntax of the expression above uses the same syntax as you would usually use when inspecting the contents of numerical data in a HIPE session, (see for instance Chapter 2) so there is no additional syntax to learn.

- **Query Example**

```
query = Query("instrument ==HIFI and band == 1a")
# a simple query should be the default form used by most users.
```

- **AttribQuery Example**

```
query = AttribQuery(Product, -'product', \
        -'product.creator=="Me" and product.instrument="HIFI"')
```

- **MetaQuery Example**

This type of query allows to inspect any part of the meta data of the product specified in the first argument.

```
query = MetaQuery(HifiCalProduct, -'h', -'h.meta["key1"].value < 123 and \
     h.meta["key2"].value == -"Hello world"')
```

> **Note**
>
> In order to obtain a numerical value (rather than, e.g., the string equivalent) it is necessary to stipulate that the meta key "value" is required, hence the need for the stipulation of query on 'h.meta["key1"].value' rather than 'h.meta["key1"]'

- **FullQuery Example**

A data mining query exploits the full interface of the product in question. Numeric functions defined in the basic toolbox are allowed:

```
query = FullQuery(Product, -'p', -'p.creator=="Me" and (ANY(p.spectrum.data <
2) \
        or ALL(p["myTable"]["myColumn"].data > 5)')
```

> **Note**
>
> Note that the `ANY` function used above is one of the standard numerical function provided for DP, and simply checks whether the expression provided in its argument is true for any of the elements in that argument. See the DP User's Reference Manual for more information.

A.3.6. Retrieving products from storage

The list of references obtained by our query with either the Product Browser or the command line allows to load the product back from the storage using *product* = *storage*.load(res[*index*].urn).product, where *index* is the index of the list entry to be retrieved. Following our example and assuming we still have the result res from our query1, we would retrieve and plot the first product in our list by:

```
p1 = storage.load(res[0].urn).product
```

The Table Dataset would be extracted and plotted with:

```
t1 = p1.get('Table1')
pl = PlotXY( t1['time'].data, t1['signal'].data,\
```

```
style=Style(line=Style.MARKED, symbol=Style.TRIANGLE) -)
pl.setErrorY(t1['error'].data,t1['error'].data)
```

In order to help know which index in the reference list is the one we are intere sted in without opening every product and inspecting it, we could sort the refer ence list by metadata entries. For example, to make the reference to the latest product appear last:

```
MetaComparator.sort(res, ["creationDate"])
```

This sorts the reference list by "creationDate", with oldest first. Other metada ta items, or multiple metadata items are also possible). However, beware: it cha nges the contents of the original variable, "res", rather than making a new list.

The Java "Collections" package (this must be imported into our session) can also be used for simple reference list manipulation. For example to reverse the order:

```
from java.util import Collections
Collections.reverse(res)
```

A.3.7. Deleting Products from Storage

Now we want to clean up our storage again, as this was just an exercise. In theory we could go into the relevant directory, identify the products by their filename and delete the respective *FITS* files. After that we would need to re-build the index. This would work for the *Local Store*, we used in our example, but in other implementations like the *DbPool* that would not be an option.

To remove our test products within the *PAL* context, we first need to identify them again by obtaining their URNs and use the method .remove() on the storage. In our example we can remove the first two items in our list as follows:

```
query1 = Query(creator == ThatsMe)

  res = storage.select(query1) storage.remove(res[0].urn)

  storage.remove(res[1].urn)
```

We can verify now with:

```
print storage.select(query1)
```

Trying to remove the third product in the previous list will result in an error, as we have no write permission to the pool test through this storage. We will need to access this pool through the other storage which was created by registering test as the first pool.

```
res1  = otherStorage.select(query1)
otherStorage.remove(res1[0].urn)
print storage.select(query1)
print otherStorage.select(query1)
```

The last two statements verify that the operation was successful and affected both storages because the pool test is registered in both. Both queries result in an empty list.

A.3.8. Updating/Repairing Storage

If the storage index becomes inconsistent, for example in the case of files being deleted or added in the directory, the index can be re-built using *pool.rebuildIndex()*, where *pool* is a pool reference obtained from the pool manager as shown above. For example the index of *Pool1* can be rebuilt with:

```
pool1.rebuildIndex()
```

There should be no attempt to access this pool during the operation, which can take a while depending on pool size.

A.4. DbPool

Used to access Products stored in a remote object (Versant) database. Here's an example:

```
# Access to Products from the default
# object database of logical name
# -'hcss.test.database'.
pool = DbPool.getInstance()
# Access to Products from an
# object database of logical
# name -'hifi.test.database'.
pool = DbPool.getInstance("hifi.test.database")
```

Note that this is an early implementation that needs to be tested thoroughly, so it is recommended to use DbPools only around test databases, or databases that are used for casual development purposes such that if data is lost, it is not a big problem.

It is recommended for performance purposes to cache products locally. To do this, wrap a CachedPool around a DbPool as follows:

```
pool = CachedPool(DbPool.getInstance())
```

A.5. HsaReadPool

The HSA read pool is an implementation that allows you to access and download observations held in the Herschel Science Archives. By default, the whole observation context is downloaded when using this pool (level 0, 0.5, 1 and 2, plus auxiliary products):

```
archive = HsaReaPool()
store = ProductStorage(archive)
```

A.6. CachedPool

The cached pool is an implementation that allows you to cache everything (including queries and their results!) retrieved from any remote pool. Any remote pool, regardless of whether it is an Oracle, Versant or whatever implementation, can therefore be cached as follows:

```
pool = CachedPool(remotePool)
```

Registering a cached remote pool allows you to work offline.

A.7. Setting up and Accessing Remote Pools

A.7.1. PoolDaemon

If you have a pool that you wish to share with someone then you can start a PoolDaemon that allows a person access and indicates whether they have read/write/query access. The PoolDaemon can be started from a command line in your system.

```
java herschel.ia.pal.pool.serial.PoolDaemon [<hostPort>(=4444)
[<poolname>(=${hcss.ia.pal.defaultpool}=stdprod)
[<loadAccess>(=true) [<saveAccess>(=true)]]]]
Examples:
```

```
java herschel.ia.pal.pool.serial.PoolDaemon
java herschel.ia.pal.pool.serial.PoolDaemon 4567
java herschel.ia.pal.pool.serial.PoolDaemon 4567 stdprod
java herschel.ia.pal.pool.serial.PoolDaemon 4567 stdprod true true
```

This makes the pool available on port number 4567.

A.7.2. Accessing Remote Pools Using the SerialClientPool

SerialClientPool (prototype) and *PoolDaemon* can be used to access remote pools.

SerialClientPool can be used for accessing a remote product pool. Usage:

```
# A PoolDaemon is running at
#    host=the.host.domain
#    port=4567
#    pool.id=foo
# create a store and register the pool:
store=ProductStorage()
store.register(SerialClientPool("the.host.domain",4567,"foo"))
```

A simple mechanism to allow read/write/query access to remote pools. This remote pool can be a Versant one (making happy all those who cannot run a Versant client such as the MacOS X fellows, or those who do not have a Versant licence), or a local store of a colleague.

Note that wrapping it up in a CachedPool ensures that you do not have to download a product twice.

A.8. More on querying

A.8.1. Querying strategy

Typically an `AttribQuery` is faster than a `MetaQuery` which is in turn faster than a `FullQuery`. Depending on the product pools that are registered, a query can take some time; to avoid unnecessary waiting time one can adopt a strategy of staged queries.

For example, a query on attributes is executed first. If too many hits are found, you can refine your query by executing another query *using the hits returned from the previous query*. This process can be repeated until the number of hits have been reduced to a reasonable amount:

```
results=storage.select(AttribQuery(...))       # 1000 hits
results=storage.select(MetaQuery(...),results) # 100 hits
results=storage.select(MetaQuery(...),results) # 50 hits
results=storage.select(FullQuery(...),results) # 3 hits
```

A.8.2. Querying for metadata in products

One thing you need to watch out when performing a meta or full query, is when you try to query for a metadata that does not exist in one or more products that you are applying the query to.

For example, consider the following `MetaQuery`:

```
query = MetaQuery(Product, -'p', -'p.meta["temperature"].value==10)
resultset = storage.select(query)
```

The query first starts creating a shortlist of all products in the storage matching type `Product`. It then runs the query string on each product in that shortlist. If any of those products don't contain the information referenced in the query string, an error is raised.

There are two ways to avoid this:

- Be as specific as you can when it comes to specifying the product type in a query. If you know the product type you want to query is of type `CalHrsQDCFull`, then specify that. Running queries using the most general product type of `Product` is not recommended, unless the products you have saved are of this type only.

- Run a two-stage query, using the `containsKey()` operator to check whether a component exists first. For example, first get a sub-set of products that contain the metadata 'temperature':

```
queryOne = MetaQuery(Product, -'p', -'p.meta.containsKey("temperature")')
resultsetOne = storage.select(queryOne)
```

Then run the original query on this subset:

```
queryTwo = MetaQuery(Product, -'p', -'p.meta["temperature"].value==10)
resultsetTwo = storage.select(queryTwo, resultSetOne)
```

A.9. Special Imports into Pools

We can import/store files of various types in pools. Here, we give some specific examples.

A.9.1. Putting a Directory of FITS Files Into a Pool

It is possible to take any set of FITS files (e.g. from the Herschel Science Archive) and place these into a pool. We can iteratively place all FITS files from a directory into a pool which can be accessed via a browser and queried using the mechanisms described in this chapter.

```
from java.io import File

lstore = LocalStoreFactory.getStore("newdir") # or any local store name
storage = ProductStorage()
storage.register(lstore)

lstore.ingest(File("C:/testdata/"), 0) # or any directory name

# To look at what you have use the Product Browser
a = browseProduct(storage)
```

In the above example a local store is placed in the default area (.hcss directory under the user's home directory) of the user's computer. It is directly accessible in the same way as other pools from there. This method does, however, not reproduce any hierarchy to the pool. It is a "flat" pool.

A.9.2. Placing Image (PNG) Files in a Pool and/or FITS File

Image data can be stored in a pool by placement in a Product with a suitable name, and saving this product in pool or in a FITS file:

```
# Obtain bytes from PNG image
bytes = -...

# Create a product with PNG data
p = Product()
p["png"] = ArrayDataset(bytes)

# Save it in a PAL pool
pool = PoolManager.getPool("myPool")
storage = ProductStorage()
storage.register(pool)
```

```
storage.save(p)

# Save it directly in a FITS file
fits = FitsArchive()
fits.save("myPng.fits", p)
```

The image can be placed in a byte array for storage in a dataset that can be placed in the pool.

```
# Obtain bytes from PNG image
# (it depends on how you generate the PNG image of a plot)
from java.awt.image import BufferedImage
from java.io import ByteArrayOutputStream
from javax.image import ImageIO

image = BufferedImage(<image name>) # implementing java.awt.image.RenderedImage
stream = ByteArrayOutputStream()
ImageIO.write(image, -"png", stream)
bytes = Byte1d(stream.toByteArray())
```

A.10. Context Products

Contexts are special types of products that contain references to other products stored.

This enables a means of building complex data structures in a storage.

There are two standard types of context products provided: `ListContext` (for grouping products into sequences or lists) and `MapContext` (for grouping products into containers with access to each by key).

A.11. Deep Copy or Cloning of Products

Say you had a context in one storage that referenced another product, and you wanted to copy that data tree to a different storage. How would you do that?

It is possible to do this using the usual `ProductStorage.save()` method. If you pass as an argument the context pointing to the 'head' of the data tree you want to clone, the whole data tree is cloned.

So for example, we have create a context with a child and store it in storageA:

```
l=ListContext()
p=Product()
l.refs.add(ProductRef(p))

storageA.save(l)
```

then we want to copy the context and child to a new storage, say storageB, all we do is as follows:

```
storageB.save(l)
```

The above cloning operation has one proviso: if a product within the data tree already exists in the destination product storage, it is not copied. A product can exist in the destination storage if for example, the original and destination storage happen to share a pool, and one of the products in the data tree being copied is in that common pool.

Note that a context may have older versions of it stored in a storage (a older version of a context may be saved when a context is saved, modified, then saved again). The older versions of the context

specified in the ProductStorage.save() argument are also cloned (if that context has any decendents that are contexts, the local versions of those descendent contexts are not cloned, however).

A.12. Common Problems

Why do I keep getting 'IndexError' or 'IllegalArgumentException: <query> could not be evaluated correctly' messages when I run my query on my PAL Product Storage?

You could get these message for one of the following reasons:

1. Your query string (the third argument of a query, eg 'p.creator==..') is simply not consistent with the jython syntax and could not be correctly interpreted by the internal jython interpreter the PAL uses. Check your query string by evaluating it on the jython command line. If your query uses a 'handle' to a product (eg the 'p' in a query 'p.meta[..]' is a handle), then create a dummy product of the type you want to query on the command line to test the query against.

2. It could be possible that the query references some data that does not exist in *any* of the products that match the product type you have passed in that query. If you see in the details of the error message something along the lines of '<something> does not exist', then this may be the case for you.

   For example, consider the following MetaQuery:

   ```
   query =MetaQuery(Product, -'p', -'p.meta["temperature"].value==10)
   resultset=storage.select(query)
   ```

   The query first starts creating a shortlist of all products in the storage matching type 'Product'. It then runs the query string on each product in that shortlist. If any of those products don't contain the information referenced in the query string, an error is raised.

   There are two ways to avoid this:

   • Be as specific as you can when it comes to specifying the product type in a query. If you know the product type you want to query is of type 'CalHrsQDCFull', then specify that. Running queries using the most general product type of 'Product' is *not* recommended.

   • Run a two-stage query, using the containsKey() operator to check whether a component exists first, e.g.

   ```
   # Get a sub-set of products that contain the metadata -'temperature'
   queryOne= MetaQuery(Product, -'p', -'p.meta.containsKey("temperature")')
   resultsetOne = storage.select(queryOne)
   # Run the original query on this subset
   queryTwo =MetaQuery(Product, -'p', -'p.meta["temperature"].value==10)
   resultsetTwo = storage.select(queryTwo, resultSetOne)
   ```

Accessing the Results of a Query

The results set can be accessed in the following way

```
a = resultsetTwo.toArray()[0].product
b = resultsetTwo.toArray()[1].product
```

Why is my PAL query so slow?

One of the possible reasons is that you are executing a FullQuery, and full queries by their very nature are the most intense of queries and are therefore the slowest.

FullQuery executions should be run as the last stage of a multi-stage query operation. Below is an example of how to search a storage for products of type 'MyProduct' that are created by a developer called 'timo', but contain a specific value in the product data itself.

```
# Stage one: Find all products of type MyProduct with creator -'timo'
attquery = AttQuery(MyProduct, -'p', p.creator=='timo')
resultset = storage.select(attquery)
# Final stage: Find all products in selection generated from previous queries,
# that has a value 10 in the column -'mycolumn' in dataset -'mydataset'
fullquery = FullQuery(Product, -'p', -'p["mydataset"]["mycolumn"].data[5]==10')
storage.select(fullquery, resultset)
```

There can be as many intermediate queries between the first stage and final stage involving AttribQuery or MetaQuery, but FullQuery's should be left to last.

A.13. Storage Product Versioning

A.13.1. Versioning

To save a set of versions of a particular edition of a Product:

```
edition = Product()
storage.save(edition) # version 0 of Product saved
# Modify edition
storage.save(edition) # version 1 of Product saved
```

To get the latest version of the Product edition, or the list of versions for that edition, you need to have available at least one, arbitrary, version. With this, you can recover the latest version of that Product, and the list of all versions of the Product in the storage. For example:

```
latest=storage.getHead(productRefOfAnyVersionOfEdition)

versions=storage.getVersions(productRefOfAnyVersionOfEdition)
```

You can get information on the current version of each product, as well as tag information, as follows:

```
print storage.versioningInfo
```

A.13.2. Querying Product Versions

The default query is to search for just the latest version of a Product edition:

```
query=AttribQuery(Product, -"p", -"1")
storage.select(query) # Just the latest versions
```

If you want to get all versions of editions that match a query, use the extended query constructors, setting the fourth argument to true (or 1):

```
query=AttribQuery(Product, -"p", -"1", 1)
storage.select(query) # All versions of Product editions that match
```

(Note that with this extended query, the special products containing versioning information, VersionTrackProduct and TagsProduct, are also returned if they match the query.)

Warning: make sure that you use the meta.containsKey() checks when performing Full or Meta-data queries, as the presence of versioned products may affect those queries, or worse, result in an exception if the metadata being queried for is not present in any product version.

A.13.3. Tagging Products in a Store

To save a product with a given tag:

```
storage.saveAs(myproduct, -"mytag")
# saves myproduct to URN=product:123, and links tag -'mytag' to that URN
storage.load("mytag")
# returns a ProductRef to product at URN=product:123
```

To assign a tag to an existing product in the storage:

```
storage.setTag("mytag", urn)
```

You can assign multiple tags to the same product:

```
storage.setTag("mytag1", urn)
storage.setTag("mytag2", urn)
storage.setTag("mytag3", urn)
```

You can re-assign tags from one product to another:

```
storage.setTag("mytag", urn1)
storage.setTag("mytag", urn2)
```

Note that the above steps removes the tag mytag from urn1, and re-assigns it to urn2. A given tag maps to only one URN.

You can also remove tags from the system:

```
storage.removeTag("mytag")
```

And check if a given tag exists:

```
print storage.tagExists("mytag")
```

A.13.4. Turning Off Product Versioning

If Product versioning is not wanted or required, you can turn off the use of versioning within your session by using.

```
hcss.ia.pal.version = none
```

A.13.5. Using the New Versioning Mechanism Against Existing Pools

You can use the new versioning mechanism against pools with previously existing data. Although it is highly recommended to use the mechanism against new pools with no data.

If you wish to use the mechanism against pools with existing data be aware that existing products in your pool do not have versioning information. So if you modify such products, and then save them:

```
p = oldstorage.load("myurn").product
// modify p
```

```
oldstorage.save(p)
```

The PAL does not know what version the modified product belongs to, and therefore saves the modified version of the product as the first version of a whole new version track.

It is therefore recommended to use the new versioning mechanism against a clean ProductStorage, devoid of any products, or as the next best thing, migrate your products to a fresh pool as follows:

```
storage.register(newpool)
storage.register(oldpool)
p = storage.load("urn:123").product
storage.save(p) # saves the product with versioning information, to newpool
```

And then use the newpool for future sessions (archive or remove oldpool).

Note also that a tool for copying pools, which reads all products and saves them back again, by preserving their hierarchy, will be placed in the HCSS at a later date. This will allow migration from old to new pools to be done more easily.

A.14. The Product Browser

The *Product Browser* was the first graphical application developed to simplify the retrieval and analysis of Products from storages.

Warning

If you are working in HIPE, we recommend you use the *Product Browser perspective* instead (Window → Show Perspectives → Product Browser).

To start the Product Browser to analyse the contents of a local store called `myLocal`, open a HIPE session and execute the following script:

```
storage=ProductStorage()
pool = LocalStoreFactory.getStore("myLocal")
storage.register(pool)
result = browseProduct(storage)
# Use the popped up GUI to explore and select products.
# The result variable will not be populated until you push
# either -'Ok' or -'Apply' in the Product Browser.
print result
```

Note

Alternatively, execute the script `herschel/ia/pal/browser/browserStart.py`

A.14.1. A visual tour of the browser

The following image shows the product browser user interface. The screen is divided into four areas:

1. The *query area*, where you enter query parameters.

2. The *result area*, where you view the query result.

3. The *result inspection area*, where you inspect a selected product.

4. The *JIDE basket area* (named after JIDE, a precursor to HIPE), where you collect the products to be returned to HIPE.
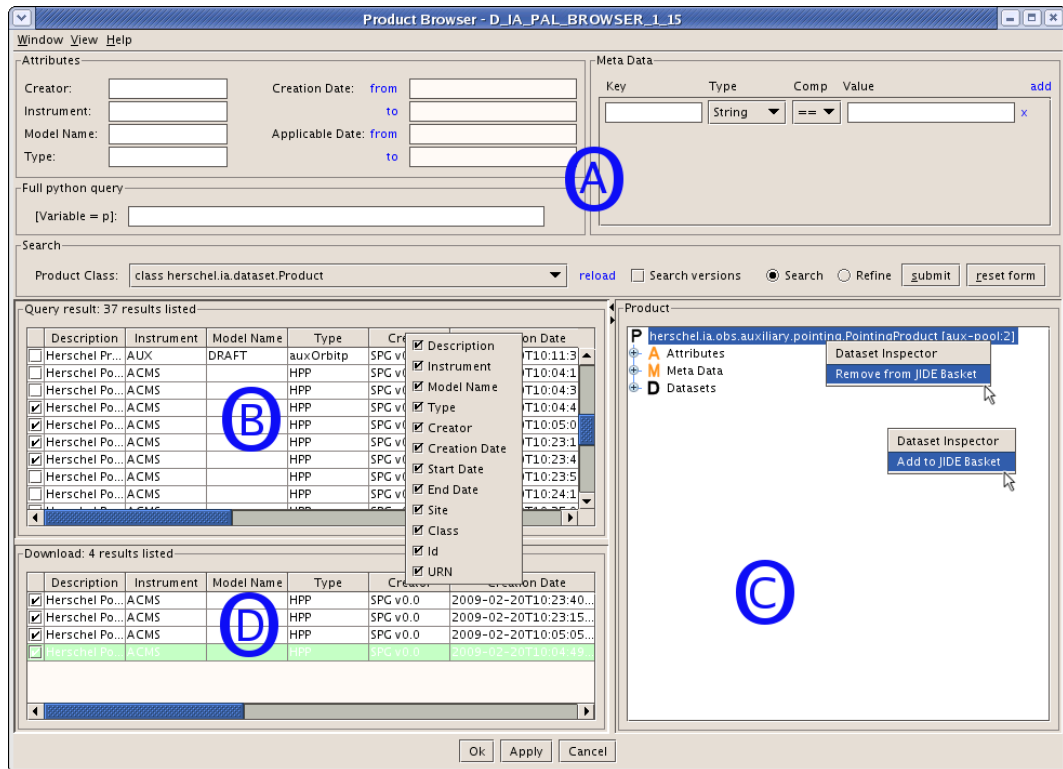
Figure A.1. The Product Browser

The following sections describe first a typical use case and then each area in more details.

A.14.2. Simple use case

1. Specify attributes of a product in the query area (A).

2. Click on the "submit" button to execute the query.

3. Review the results in the result area (B).

4. Optional: if there are too many results, refine the query by specifying values in the *Attributes* and *Meta Data* panes, and/or a query in the *Full python query* pane. Then select the Refine radio button and press submit again.

5. Inspect selected results in the result inspection area (C).

6. Transfer the results of interest to the JIDE basket from the area (B), by marking the checkbox at the beginning of the corresponding row, and (C), by right-clicking and selecting Add to JIDE Basket from the menu.

7. Click Ok or Apply and process the selected results in HIPE. The results are available in the return variable of the `browseProduct` method (in the browser start example above it is called `result`).

A.14.3. A: Query area

The query area is divided into three input panes: *Attributes*, *Meta Data*, and *Full python query*.

1. Attributes queries search commonly defined attributes only.

2. Meta data queries search on additional meta data specific to a product. You need detailed knowledge about a product to specify meta data queries. However, the result inspection area (C) may be used to see available meta data for a product.

3. Full python queries allow to specify free form queries in the Jython query language. Refer to the documentation of the `ProductStorage` class for further information on this topic.

Note that all attributes and meta data parameters are joined by the AND operator.

Note for power users: for simple OR-queries you can use the JIDE basket (D). First, do a query for the first term (for instance, *Creator = André*) and add the results to the JIDE basket. Then, do a query for the second term (for instance, *Creator = Marc*) and add the results to the JIDE basket.

For more complex OR-queries you can use full Python queries, although they might become very slow. Complex OR-queries on meta data level are currently not supported.

A.14.4. B: Result area

This table displays all products that match a specific query.

Check or uncheck a product to move it to or remove it from the JIDE basket.

You have several possibilities to rearrange the products:

- Click on a column header to sort rows in ascending or descending order.

- Right click on a table header to pop up a context menu where you can select which columns to show (shown in Figure A.1).

- Drag and drop a column header to rearrange the column order.

- Click between two column headers to resize a column.

Settings are stored between sessions. To revert to default settings, choose Reset User Preferences from the Window menu. You will have to close and restart the browser for the change to take effect.

You can choose the column layout for the *Query result* pane from the Table Layout entry of the View menu. Two predefined layouts are available:

- The *Default Table Layout* includes the following columns:

  - Description (Attribute): self-explanatory.

  - Instrument (Attribute): self-explainatory.

  - Model Name (Attribute): self-explanatory.

  - Type (Attribute): self-explanatory.

  - Creator (Attribute): self-explanatory.

  - Creation Date (Attribute): self-explanatory.

  - Start Date (Attribute): self-explanatory.

  - End Date (Attribute): self-explanatory.

  - Site: the data store of the result.

  - Class: the class of the Product as encoded in the URN.

  - Id: the unique id within the data store.

  - URN: convenience column for copy & paste. If you triple click into a cell of this column you can select and copy the URN to your operating system clipboard. This is one way to use the browser independently from HIPE.

- The *ObservationContext Layout* lists data of interest to astronomers from an *Observation Context*, that is, a Context containing all the Products related to an observation. The layout contains the following columns, which should all be self-explanatory:

  - Observation ID.

  - Operational day number.

  - Observation start time.

  - AOT ID.

  - Instrument mode.

  - Target name.

  - Proposal ID.

  - AOR label.

See Section A.14.7 for instructions on how to add a new layout (for advanced users only).

A.14.5. C: Result inspection area

Select any entry in the query result area (B) or in the JIDE basket (D) to inspect its attributes, meta data and children in the result inspection area C. The selected product or context will be displayed in a hierarchical tree structure.

There are five types of nodes:

- **P:** a Product contains the real data and can be examined with the dataset inspector. To open the dataset inspector you can either double or right click on the node.

- **C:** a Context contains other Contexts or Products.

- **A:** a predefined set of Attributes common to all products and contexts.

- **M:** Meta data that is specific to each type of products.

- **V:** old Versions of a product or context.

To add or remove products and contexts to or from the JIDE basket you can right click and select the appropriate action from the menu (both entries are shown in Figure A.1.

First note for power users: The current implementation of the tree supports only contexts that are inherited from `ListContext` or `MapContext`. This is due to missing generic meta information about the children of an ordinary context.

A.14.6. D: JIDE basket area

The JIDE basket collects the products and contexts of interest. Clicking on Ok or Apply will make the content of the basket available within HIPE. Ok will close the browser, Apply will keep it open for further usage. Note that the results are sorted the same way as in the JIDE basket.

Now you can further analyse the results in HIPE. Note that the ProductBrowser will return a list of ProductRefs rather than a list of Products. A ProductRef is a small object that stores a pointer to a Product, without loading the Product into memory.

```
result = browseProduct(storage)
# This will print the list of ProductRefs
```

```
print result
# This will print the first ProductRef in the list.
print result[0]
# This will print the first Product in the list.
print result[0].getProduct()
```

A.14.7. Advanced: Adding a Table Layout

These instructions show you how to add a new entry to the Table Layout submenu of the View menu.

You can use either Jython or Java to register new Table Layouts. You can even add a new set *while* the browser is running and you can define your own columns as long as you extend from `AbstractBasketColumn`.

The Java code to register a Table Layout is shown below:

```
TableLayoutRegistry.instance().registerTableLayoutBuilder(new TableLayoutBuilder() {

  public void configureBasketTableModel(BasketTableModel model) {
    model.addColumn(new AttributeColumn(model,
AttributeColumn.ProductAttribute.DESCRIPTION));
    model.addColumn(new MetadataColumn(model, -"test", -"test label",
String.class));
  -}

  public String getId() {
    return -"simpletablelayout";
  -}

  public String getLabel() {
    return -"Simple Table Layout";
  -}

});
```

Appendix B. Using JIDE or the JIDE view in HIPE

B.1. Introduction

A DP session involving scripting is typically initiated within a console window of HIPE or JIDE. This window includes help and history for the session. Individual commands can be input to the console using DP/Jython commanding, which is discussed later in this chapter. Alternately, the console and associated editor window allow for the construction and running of complete algorithms based on the Jython language or even sections/individual lines of algorithms. Since no separate compilation is required, individual lines or sections of algorithms can be checked for validity very quickly. DP scripts that use GUIs can also be started from within the HIPE/JIDE view. *Example HIPE/JIDE input code is provided throughout the text in shaded boxes. Comments on the code and, frequently, example output are provided within the boxes on lines preceded by the "#" mark.*

In this chapter we discuss how to start working in the DP console view of HIPE or JIDE. We provide some simple DP interactions to illustrate its use. We discuss some more detailed scripting capabilities in Chapter 1.

B.2. Scripting using the JIDE view of HIPE

For an introduction to HIPE see the *HIPE Owner's Guide*.

In this section we describe the *Classic (JIDE)* perspective of HIPE, which gives access to a legacy graphical interface called JIDE.

From the HIPE Welcome window select the Classic (JIDE) icon at top right of the screen ( ).

Alternatively, select Window → Show Perspectives → Classic (JIDE), as shown in the following figure:

Figure B.1. Selecting the Classic(JIDE) perspective in HIPE.

The JIDE perspective contains three windows: an Editor view to the top of the screen, a Console view towards the bottom left and a History view towards the bottom right:.

Figure B.2. The Classic(JIDE) perspective in HIPE.

An interactive**Console view** appears at the bottom left of the perspective with a customizable `HIPE>` prompt. Individual DP commands can be run here. Click in the Console view with your mouse, then type the following and press **Enter**:

```
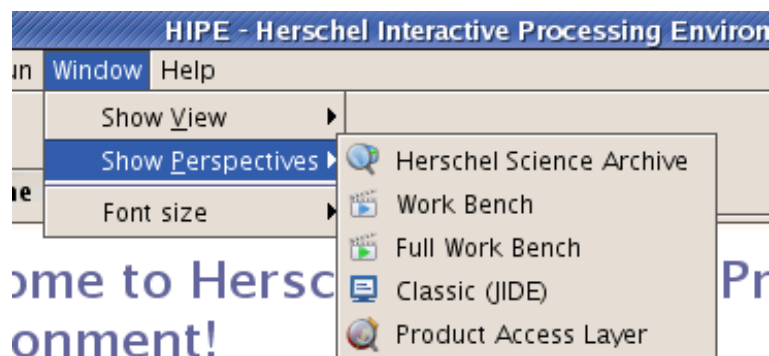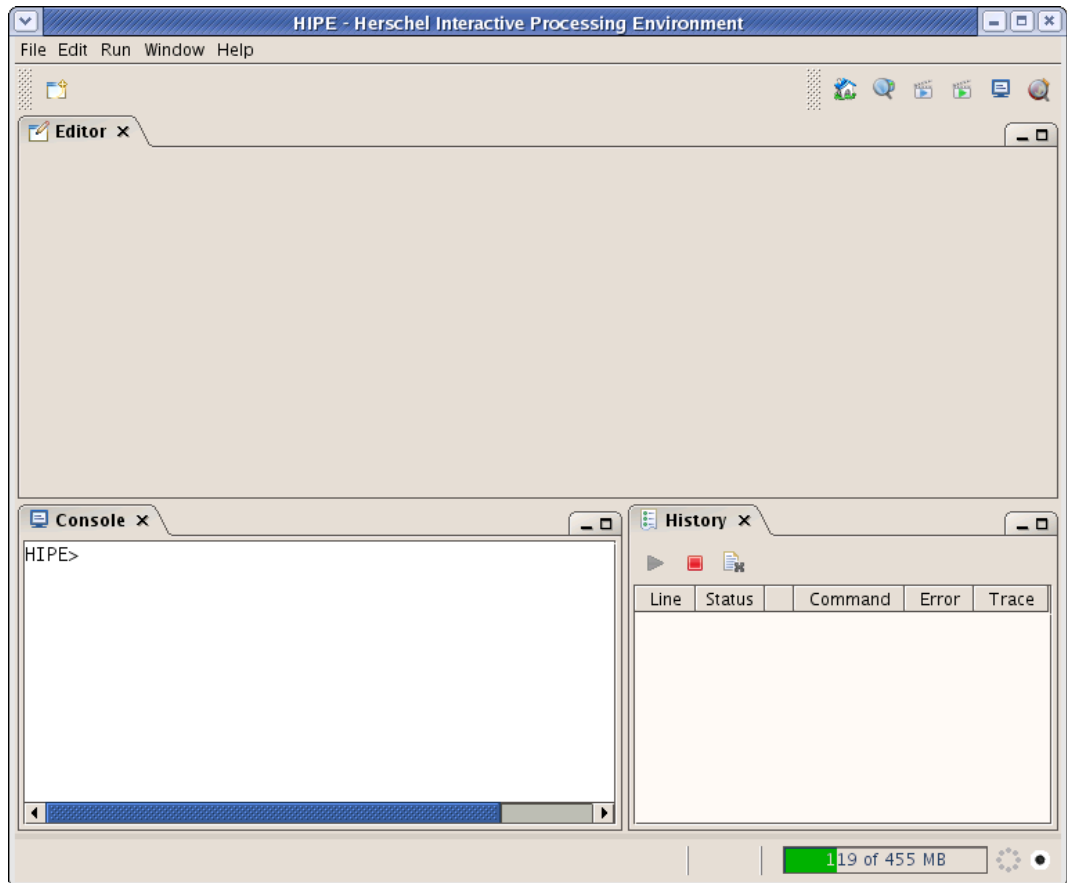print 5 + 3
```

The answer will appear on the next line, followed by the `HIPE>` prompt again:

```
HIPE> print 5 + 3
8
HIPE>
```

> **Note**
>
> In a plain Python or Jython console it would be enough to type `"5 + 3"` followed by the **Enter** key to get the result. In DP we have to use the `print` keyword, otherwise we would get no output.

The bottom right of the perspective contains a **History window** that lists the commands (including those inside algorithms) executed in the current session. A red circle with a cross in the *Status* column means that the corresponding command caused an error. Information on the error can be obtained from the *Error* and *Trace* columns.

You can try this functionality with the following command, which will generate an error:

```
sign 5
```

The top pane of the perspective is an editor for developing your scripts. To start using it, select File → New → Jython Script. This will create a blank script within the New-1 tab:

Figure B.3. The Classic (JIDE) perspective with script screen made available.

Click in this window and type in a similar print command to the above example. Pressing **Enter** will not run this simple script. To run the one line, click in the grey margin to the left of the line you have typed. An arrow should appear beside the line. Now go to the line of icons at top left of the HIPE

screen and click on the single arrow (   ). This will run your one-line algorithm and the result will appear in the lower left command line window.

After this introduction to the three windows of the HIPE Classic (JIDE) perspective we will consider each of the menu and icon items in turn.

B.2.1. File menu

Only one of the **File menu** items has an associated icon (the "Save" capability).

Use **New** creates a new window for algorithm development ("Jython script") or text ("Text file") in the top "Editor" view of HIPE (note that a new "Tool" window feature is yet to be developed).

Open File allows a file to be opened in the Editor that is chosen from anywhere within the system (ASCII - DP script files are typically stored with the suffix .py, in ASCII format). If the suffix is .py the window is always a Jython script window -- otherwise a text window.

Close closes the current window shown in the Editor view. **Close All**
closes all the windows showing in the Editor view.

Save and **Save As** for saving the current algorithm shown in the top window. The "Save" capability

is also available using the icon   shown in the line of icons to top left of the HIPE window.

Revert Reverts back to the original version of the file currently being editted.

Refresh this IS NOT FOR USE WITH THE EDITOR. This capability is for the Navigator view available in HIPE. The Navigator view automatically updates every 5 seconds so that new/changed files in the computer system (e.g., copied files) are made available in the Navigator view of HIPE. Hitting F5 or "Refresh" does this instantaneously.

Rename this IS NOT FOR USE WITH THE EDITOR. This capability is for the Navigator view available in HIPE. It allows the renaming of a highlighted file showing in the Navigator view of HIPE.

Print prints text of HIPE Editor session to a printer (various page types and setups) or postscript file.

Exit exits from the HIPE session. For any unsaved changes to any of the files showing in Editor windows the user is given the opportunity to accept or reject changes before HIPE is closed down.

B.2.2. Edit menu

Most of the Edit Menu functions (except Cut, Copy, Paste and Open) have an associated icon at the top of the HIPE panel. The associated shortcut icons are shown next to the function name in the menu. Each function also has an (standard) associated CTRL combination (except for Open and Open With). See Figure B.4.
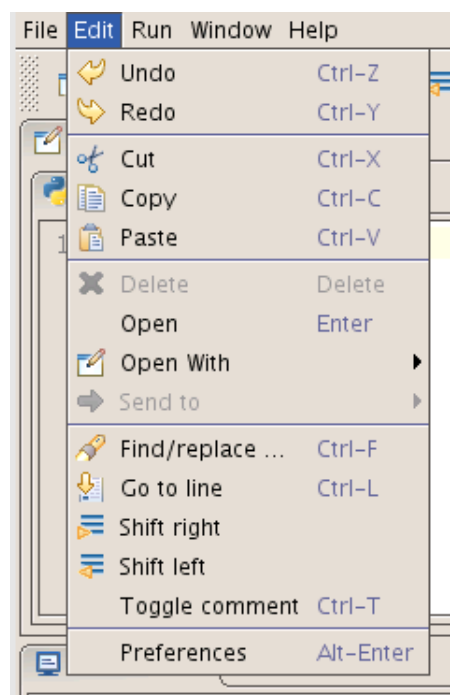
Figure B.4. The HIPE edit menu.

Undo (CTRL-Z) and Redo (CTRL-Y) and allows edits (cut/paste or deletion from the keyboard) to be undone or redone.

Cut (CTRL-X), Copy (CTRL-C) and Paste (CTRL-V) These provide the usual cut, copy and paste facilities, using the mouse to select and position text in the Editor window.

Open (enter key), Open With, and Delete (delete key) these are NOT FOR USE WITH THE EDITOR. This capability is for the Navigator view available in HIPE. It allows the highlighted file in the navigator view to be opened in the HIPE Editor view -- as Jython Script, text editor (default

for Open) or File Overview (gives size/type of file info), or delete the highlighted selection from the system.

Find/replace (CTRL-F) does the usual find and replace of text within the current window of the HIPE Editor view.

Go to Line (CTRL-L) allows the user to go to a specified line number.

B.2.3. Run menu

The Run Menu items all have associated icons at the top of the HIPE window.

Stop (ALT-T) - stops a script being executed. Click on this button or choose **Stop** from the pulldown menu to stop the execution of a script before it reaches the end. Note that this icon is greyed out when there is no script in execution.

Run (ALT-U) - runs a single line or logical block of a script. A selected set of lines can be highlighted using the mouse and these can be executed by then clicking the Run button or selecting Run from the menu. The lines are iterated to the console window and their status shown in the History window to bottom right. While running, the red stop button is lit.

Run all using pulldown or icon, this allows all DP commands in the current Editor window of HIPE to be run in sequence. The lines are iterated to the console window. The stop button turns red while running.

B.2.4. Window and Help menus

The "Window" menu allows access to HIPE perspectives (such as the Classic(JIDE) discussed here) and views. There are a number of views available which are discussed more extensively in the "DP HowTo's" document. By selecting one of the offered views an extra panel is added to your HIPE perspective. For example, in the Navigator view showing the available directories and files on your system is added in a panel to the right on the HIPE screen.

Figure B.5. Adding the Navigator view to the Classic(JIDE) perspective in HIPE.

The "Help" menu, in addition to providing access to Help inside of HIPE (together with Help search facilites) also provides "About" information on HIPE and access back to the Welcome page that you get on starting up HIPE.

B.3. DP scripting using JIDE

DP users who wish to do scripting may choose to work within DP JIDE separately from HIPE. You can start the JIDE console can be initiated from the "Start" menu after installation using the HCSS installer.

Alternately, it can be started at a command window prompt.

```
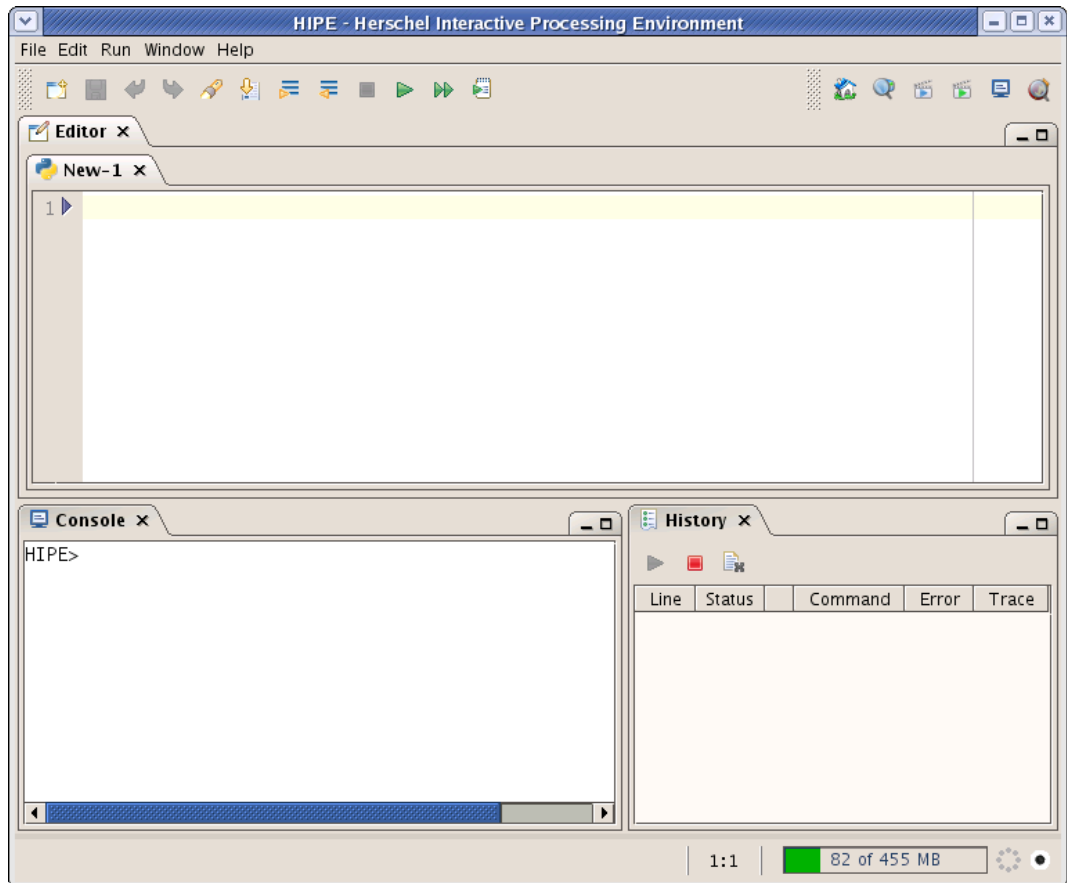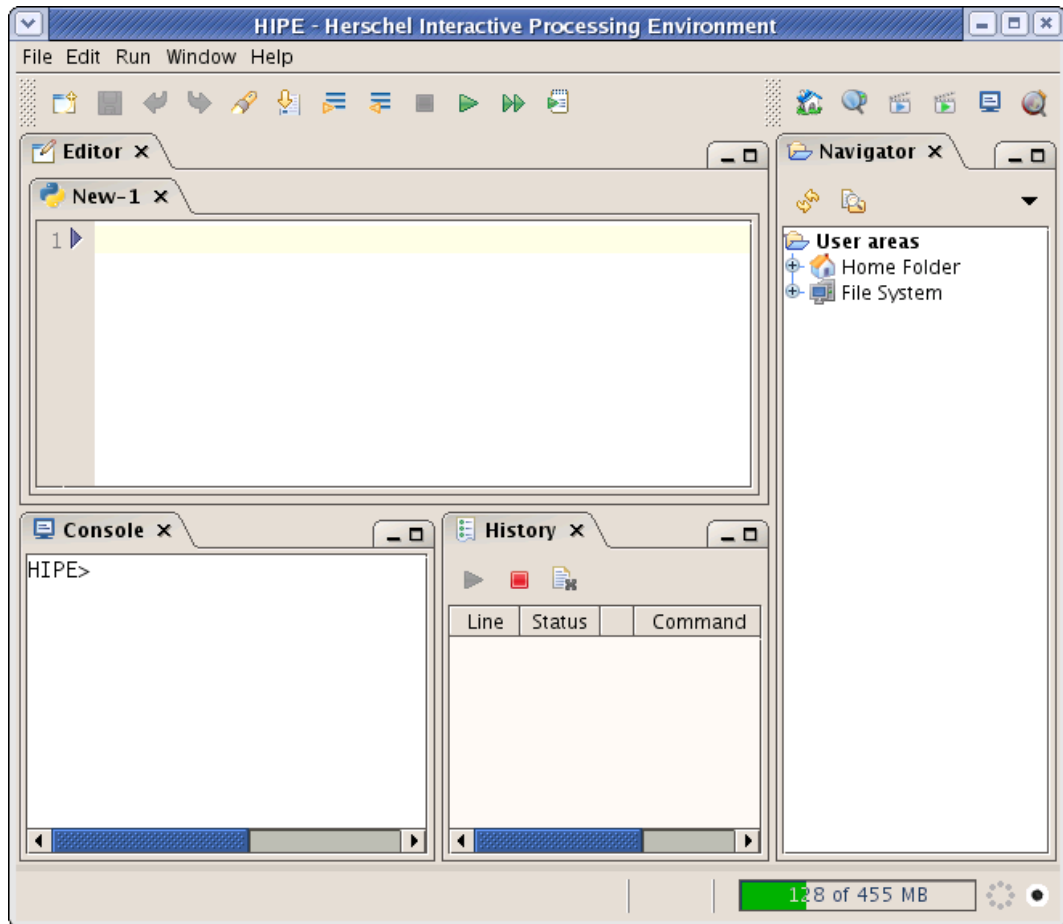$ jide
```

Note

- Under Windows, open a command window and type in the same thing, or execute `jide.bat` from the `bin` directory of your HCSS build.

- Under Mac, starting from the command line only works if you installed a developer build via the Continuous Integration System. If you used the *InstallAnywhere* installer instead, you have to start the application via its icon.

Note that some feedback from the DP session is provided to the terminal window from which it was started. This includes information on the settings used on JIDE startup and information on database access (basically feedback on where interactions occur with systems outside the immediate DP session). The JIDE shell performs the following tasks:

- Loads a customised DP environment (imports a set of libraries and defines a set of variables).

- Keeps a history of successful DP statements.

- Implements a set of basic editing functions (copy, paste, find and replace).

It is an extension of the standard Jython shell. Here, we provide some basic startup information.

If entering the JIDE command from a terminal window, information on preloaded elements in the DP session appear in the terminal window. Startup from the "Start" menu goes directly to the following. After any feedback, a separate three-paned console window should appear (see Figure B.6). The bar at the bottom of the window displays the amount of memory used by the session: as memory usage increases the bar will turn from green to yellow and then to red.

Figure B.6. The JIDE window set-up.

The JIDE window has three components. An interactive **command line/console window** is given to bottom left of the view with a customizable "IA>>" prompt. Individual DP commands can be run here. Click in the bottom left window with your mouse, then type in

```
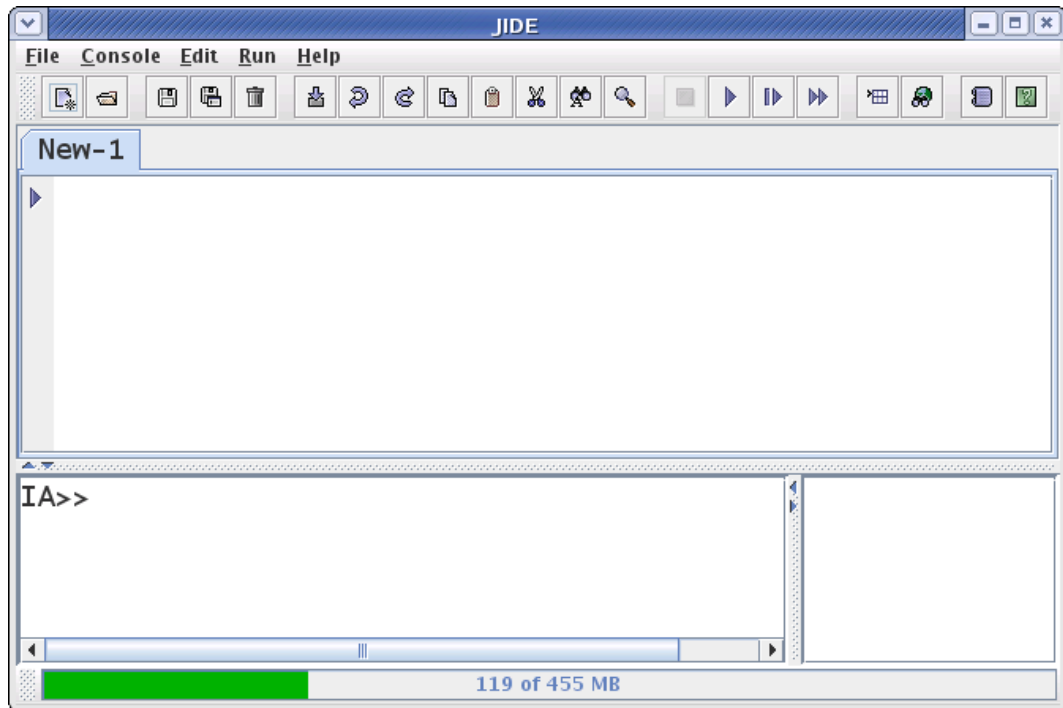print 5 + 3
```

Followed by **Enter**. The answer should be provided on the next line, prior to receiving the "IA>>" prompt back again:

```
IA>> print 5 + 3
8
IA>>
```

> **Note**
>
> In a plain Python or Jython console it would be enough to type "5 + 3" followed by the **Enter** key to get the result. In JIDE we have to use the `print` keyword, otherwise we would get no output.

The bottom right of the console contains a **command history window** that lists the commands (including those inside algorithms) used in the current session. Any command highlighted by a red cross next to it caused an error. Some information on the error that occurred can be obtained using the mouse to click on the command highlighted. A response with the error is shown in the command line window to bottom left. Try the following

```
sign 5
```

After hitting **Enter** the user will see the history window has a command highlighted by a red cross next to it. Click on this using the left button of the mouse. This then expands the information on the error incurred.

The top pane of the console is available for the user to develop his/her own algorithm using the available DP commands. Click in this window, type in a similar print command to the above example. Hitting return will not run this simple script. To run the one line, click in the grey margin to the left of the line you have typed. An arrow should appear beside the line. Now go to the line of icons and

click on the single arrow ( ). This will run your one line algorithm and the result will appear in the lower left command line window (again). If you want to "print" a string it needs to be in quotes (e.g., print "Hello World").

Note

The top pane is not meant to be a fully-fledged text editor, nor a sophisticated IDE (Integrated Development Environment). It offers basic editing and debugging capabilities for developing simple scripts, but larger projects should be developed in external tools and then loaded into JIDE for execution.

Now that we have a brief introduction to the three windows of JIDE we will consider each of the menu and icon items in turn.

B.3.1. File menu

Each of the **File menu** items has an associated icon except for exit. These are the first 5 icons on the bar under the menu headings.

New creates a new window for algorithm development. New history and/or command line windows are not created.

Open allows a file to be opened in the top window (ASCII - DP files are stored in ASCII format).

Save and **Save As** for saving the current algorithm shown in the top window.

Close closes the file in the top window pane. Only closes the window showing the current algorithm.

Print prints text of JIDE session to printer or postscript file.

Screenshot as JPG creates JPG file of screen view.

Screenshot as PNG creates PNG file of screen view.

Exit exits from the JIDE session.

B.3.2. Console menu

Execute in the console requests the input of a DP script file, loads it and runs it inside of JIDE.

Execute does a similar thing, except it runs the whole script on the system rather than using the JIDE console

Execute in the background does the same as Execute, but runs the script in the background.

Save history and **Save history as ...** saves a history of successful JIDE commands from this session.

B.3.3. Edit menu

Each of the Edit Menu functions (except Goto) has an associated icon at the top of the JIDE panel (middle section of icons).

Import history allows the import of the history of a saved JIDE session.

Undo and redo and allows edits (cut/paste or deletion from the keyboard) to be undone or redone.

Cut and paste and the usual cut and paste using the mouse to select and position text.

Find/replace does the usual find and replace of text within the upper window of the JIDE console.

Goto allows the user to go to a specified line number.

B.3.4. Run menu

The next four icons at the top of the JIDE window relate to the **Run menu**.

Script mode This only appears in the Run Menu. The default is that the script mode is disabled, the Run, Run selection and Run all buttons then work as if on the command line for lines of code written in the debug window and the commands are reiterated to the console. In script mode, only requested output (e.g., from a "print" command) will have output sent to the console.

Stop - stops a script being executed. Click on this button or choose **Stop** from the pulldown menu to stop the execution of a script before it reaches the end. Note that this icon is greyed out when there is no script in execution.

Run - runs a single line or logical block of a script. The line is iterated to the console window, unless in script mode (see under "Run Menu") when only explicit outputs from script commands appear at the console. In script mode the button turns red.

Run selection select a set of commands by dragging the mouse over them. Pull down to **Run selection** (or click the icon) to run these DP commands only. The lines are iterated to the console window, unless in script mode (see under "Run Menu") when only explicit outputs from script commands appear at the console. In script mode the button turns red.

Run all using pulldown or icon, this allows all DP commands in the top pane of JIDE to be run in sequence. The lines are iterated to the console window, unless in script mode (see under "Run Menu") when only explicit outputs from script commands appear at the console. In script mode the button turns red.

B.3.5. Help menu

The last four icons at the top of the JIDE window relate to various forms of help that are also available under the **Help** pulldown menu.

Dataset Inspection allows the user to view datasets (notably tables) currently available in the DP session in a separate `Dataset Inspector` code window (see Figure B.7). For more about the Dataset Inspector see the *Data Analysis Guide*.
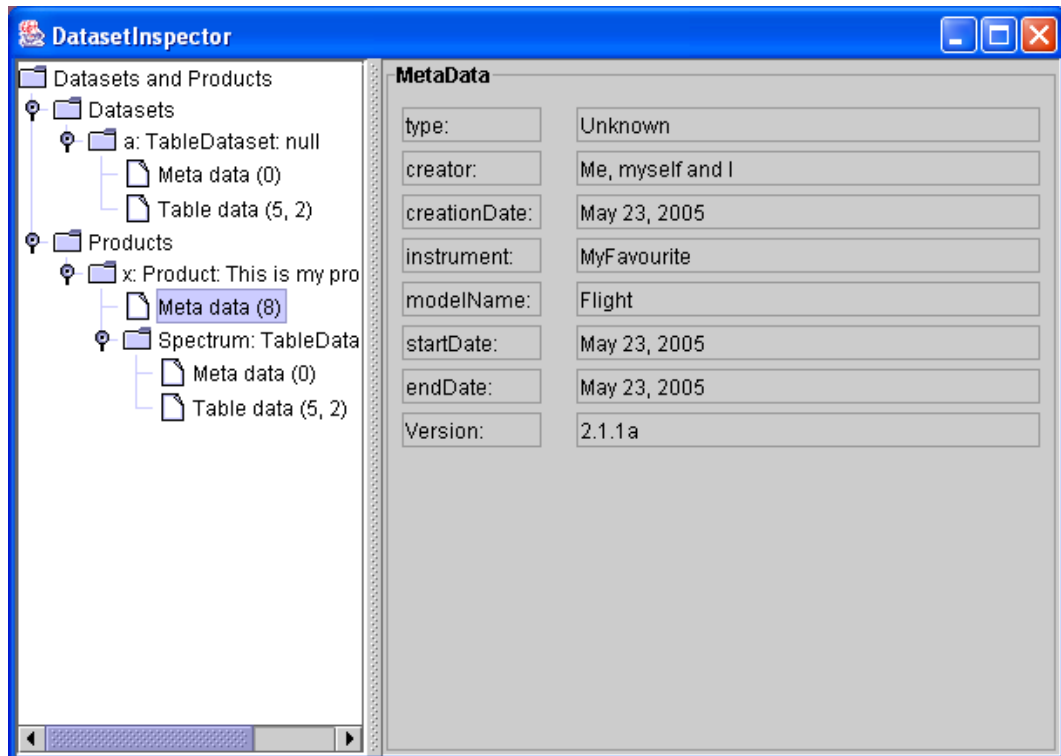
Figure B.7. The `Dataset Inspector` window

Session Inspection allows the user to view the classes (programs) and functions available in the current DP session. Also allows the user to inspect all variables used in a session. See Figure B.8. Further classes and functions can be made available by importing "packages" (see Chapter 5).
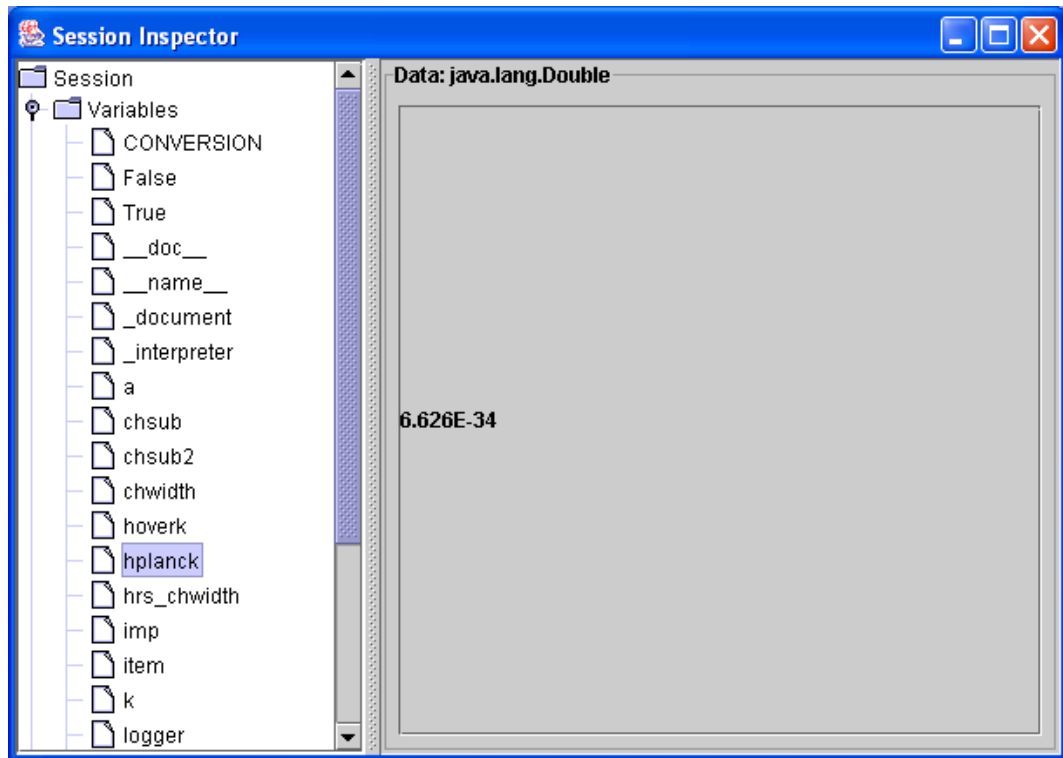
Figure B.8. The `Session Inspector` window

Log Window provides a listing of the feedback from running commands in the system, including error messages. These appear in a separate Log window. The log can be saved when exiting from JIDE.

Access to On-line Help Documentation opens the HIPE Help System in your default browser.

B.4. Quitting JIDE

We already know that the Exit entry in the File menu can be used to quit JIDE. In this case a new window appears, prompting the user to save the current work (scripts and command history). You will get a list of all unsaved files, together with entries like

- [New-1]: -no file associated-. This is a script that has not been saved yet (beware that it could be an *empty* script).

- [History of Console1]: -no file associated-. This is the history of the commands you have issued, listed in the lower right panel. Useful if you want to save to a script what you have typed.

To select an item click on it. You can select multiple items by holding **Ctrl** while clicking on them; if they are contiguous you can select them in one go by clicking on the first one and then clicking on the last one while holding **Shift**.

Below the list of unsaved items there are four buttons: Select all to select all the items, Save Selected to save the selected items, Cancel to go back to JIDE without quitting, and Close to quit JIDE.

After pressing Close, a second confirmation window is displayed. Click Yes to quit or No to go back to JIDE.

An alternative way to quit is to type `System.exit(1)` at the `IA>>` prompt and press the `Enter` key. This command can also be added to a script (for more information about writing scripts, see).

Warning

The `System.exit(1)` command causes the current JIDE session to terminate immediately. All unsaved work will be lost.

B.5. Standard settings for JIDE and HIPE

JIDE and HIPE come with a memory specification that is dependant on the installer information supplied by the user when setting up the system initially. The settings are specified in the startup script for JIDE. This script is located in the `$HCSS_DIR/bin` directory (named `jide.lax`. These settings can be modified by editing this JIDE startup script. The following two lines adjust the initial and max memory allocations.

```
lax.nl.java.option.java.heap.size.initial=134217728

lax.nl.java.option.java.heap.size.max=536870912
```

A similar `hipe.lax` file has the same editable lines. Make sure that the environment variable `HCSS_PROPS` is properly defined.

Make sure `HCSS_PROPS` contains the specification of the standard var.hcss.dir property (this should be the property defined in your `$HOME/.hcss/myconfig` file IF you have set up your own environment and are not using a local network installation or an installer). And be sure that `var.hcss.dir` points to the HCSS build directory. You can check any property with a command such as the following in the Console area.

```
print Configuration.getProperty("var.hcss.dir")
```

B.6. DP working directory and file access

The current working directory of DP is the directory from which JIDE/HIPE was started. Jython has some limitations, inherited from Java, with regard to navigation of the underlying operational system. However, changing the default directory can be accomplished in two ways.

By changing the underlying system path using *sys.path*. This can dynamically change the default directory.

```
# at the console command line type
import sys # if -"sys" not already imported
sys.path.insert(0, -'/dir/path')
# the -'0' puts it to the front of the directory path of the user.
```

By setting a standard directory in the path by putting the name of a directory in the file ".jython" under the users home directory. This then means that, from whatever directory JIDE or HIPE is started, this directory is always in the path.

But the user is advised to start JIDE/HIPE from a directory where he/she is going to read/write files by default and to use absolute paths for the file names.

When using "Save" under the File menu of JIDE/HIPE the user can specify any directory.

A view of the current directory contents is available through the HIPE navigator view. Such a view is not possible with JIDE. Opening a file in either HIPE or JIDE under the "File" menu does allow a view of the available files in any directory on the system.

It is possible to print the file contents of the current working directory using the following in a console window.

```
import os

# print the working directory
print os.getcwd()
# print the names of the files in the working directory
print os.listdir(os.getcwd())
# any directory name can be placed in the brackets
```

This provides an unsorted listing of all files and directories in the working directory. If the user wants to filter the file list, e.g. to select only the fits files, then a *glob* module can be used with search pattern following the UNIX shell rules, i.e. "*", "?", "[]" etc which are interpreted in the same way as in the UNIX shell.

```
import glob

ffiles = glob.glob("*.fits")
# or even more elaborate example to provide the list of all fits file
# in a given directory and perform some action on them
ffiles2 = glob.glob("/home/user/scratch/fitsfiles/*.fits")
fits = FitsArchive(reader = FitsArchive.STANDARD_READER)

for fi in ffiles2:
    product = fits.load(fi)
    # do something on the products, like print the dimensions
    print fi, product.default.data.dimensions
#
```

B.7. Getting command-line help

Further help in JIDE or HIPE is available through command-line interaction. There are two methods.

- help() -- which provides an overview of the help system via a separate popup window (see Figure B.9). The window also includes all documentation provided by each of the instruments, for specific aspects associated with handling instrument information, providing more specialised documentation.

- In HIPE, selection of help through any button marked .... provides access to Help that is shown in a browser. Search and full Help document selection is available through this system.

Figure B.9. The online `help()` popup window

B.8. Programming loops

Earlier in the chapter we tried some basic commands to illustrate the components of the HIPE and JIDE windows. One particular capability of JIDE and HIPE is allowing block support for DP coding. Suppose we want to take a basic print command typed in the command line window.

```
a = 5    [Enter]
print a    [Enter]
# 5
```

Now simply input (the [Enter] means you have to press the enter key on your keyboard)

```
for i in (1,2,3): [Enter]
```

This will return a `....` response in the command line. Note that the colon at the end of the line is important for starting the block. The command is incomplete. Input a `print i` command *indented by at least one space*. A further `....` is returned. Hit **Enter** once more, the command is now complete.

The whole session should look like (again, note the indent prior to the print statement on line 2):

```
for i in (1,2,3):
.... print i
....
#1
#2
#3
```

We could have added a number of commands to this `for` loop. The block statement continues until a blank line is produced. The history of the window is now available. The up arrow will provide the previous command, which can then be edited as desired and re-entered

```
for i in (1,2,3):
  print i
```

You can edit this block statement in the bottom left panel of JIDE by using the LEFT and RIGHT keys (not UP and DOWN, these are used to move through the history) and deleting/adding characters.

Blocks within blocks (nested `for` loops or `if` statements) are also possible. Basic rules about the use of blocks follow Jython language syntax.

- Each statement in a block must begin in the same column;

- Each of the DP key statements and clauses (**class, def, for/else, if/elsif/else, try/except/else, try/ finally and while/else**) denotes the beginning of a new block;

- A new block must be indented at least one space from the enclosing block;

- The end of a block is marked by having the next statement begin in the same column as the enclosing blocks.

For example

```
for x in (1,2,3):
  print x # outer block
  for y in (4,5,6):
    if y == 5: # inner block
      print y # inner-inner block
    print x*y # inner block
    # insert inner block statement here
  # insert outer block statement here
```

As usual, end with a blank line! Note the end of each `for` loop is determined by where the indentation ends.

B.8.1. Loop performance on arrays

Numeric Arrays are discussed in Chapter 4 of this manual. But we mention here how loops can be computationally expensive when used with numeric arrays in the system.

In performance checks using the HCSS timing differences for standard operations (e.g., division and multiplication many times on arrays) are found to be very similar to using similar programming languages such as Python. However, Jython/HCSS loops can be slow and for large computations this can become very inefficient for the user.

One means of reducing quite significantly the computation time for simple arithmetic computations on arrays is to use the ability of the HCSS language to do in-line calculations. For example:

```
z=Double1d(x.size)  # create a 1d numeric array of the same size as an original
                    # array called -"x"
for i in range(1000):
    z.set(x)   # assign, not allocate
    z-=y       # inline subtraction
    z/=c       # inline division

# instead of the following --- which is much slower
for i in range(1000):
    z = (x-y)/c
```

For large operations this can reduce computation time by nearly an order of magnitude.

Some further advanced tips to improve performance are provided in [???](#).

B.8.2. Using the Editor view with loops

The top edit window of JIDE and Editor view of HIPE can be used to keep lines of code in your session. To run things in this window we have three "arrows" at the top of the JIDE screen (two in HIPE). The single arrow on the left of these will run things as if you were putting them on the command line. So if we have a "for" loop a blank line will stop the loop. However the middle arrow (runs a highlighted section of code -- incorporated into single arrow also for HIPE) and the double arrow (which runs everything within the currently opened edit window) run commands within the whole group in the editor window sand ignores blanks. For example, we may consider the following lines of code.

```
for i in range(4):
  if i > 0:
    print i

  j=i
  print j-i
print -"Finished"
```

If run line-by-line (mouseclick to produce arrow next to the "for i in range(4)" line -- then hit the single arrow at the top of JIDE or HIPE) then only the first loop is run before a blank line is encountered. If the double arrow is used then the blank is ignored and the whole thing is run.

> **Warning**
>
> This means that the way blank lines are treated depends on how the DP code is run. Your code will run differently if you run it line-by-line as compared to running it as a complete script.

B.9. Multiline statements in the console view

Another improvement of JIDE/HIPE compared to other Jython interpreters is that it allows multiline statements. The backslash (\) character at the end prevents execution of the line when hitting Enter and the statement can be continued.

The following example breaks up a longer definition of a tuple into three lines:

```
IA>> a = ("meaning", -"of", -"life", \
.... -"shrubbery", -"killer rabbit", \
.... -"holy hand granade")
IA>> print a
('meaning', -'of', -'life', -'shrubbery', -'killer rabbit', -'holy hand granade')
IA>>
```

Note that the backslash initiates a continuation mode. The mode is left upon hitting Enter after the first line without backslash, and the entire line is executed.

B.10. Pausing during script execution and debugging in JIDE and HIPE

A script may be paused at any point using the `pause()` command. This allows values to be changed while a script is paused in the Debug window.

See the following example script.

```
from  herschel.ia.jconsole.util import * # import pause
def test(arg=1):
 a=12
 for i in range(arg):
  pause() # pause here, change of a within the debugger is allowed!
  a=a+i
  print a
  pause() # and  here
  print a

test(5) # run the example
```

Once you execute the above script, the following window will pop up.
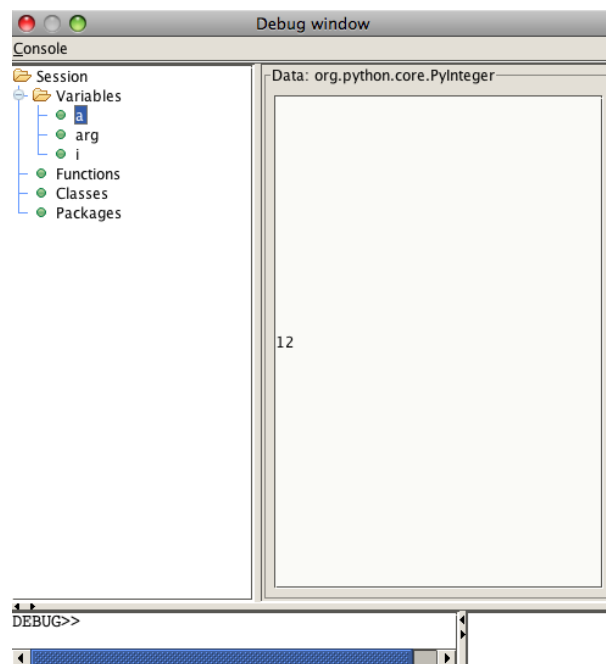
Figure B.10. The Debug window

You can change the value of the required variable by setting it via the DEBUG command line, and can be checked by clicking on the variable name in the main window. Once the change has been made in the Debug window to the required variables, select the "Console" pull-down menu in the Debug window. To continue to execute the scroll down to "Resume". This will allow you to exit the Debug window and to continue to execute your script.

B.11. Background script execution

There are two ways to run time consuming scripts in background. One is from the drop-down menu under "Console" -> "Execute in background" which executes, in the background, the script which is loaded in the JIDE editor window. This is not available in HIPE.

The other method is by using the `execfile` capability, e.g., `bg('execfile("script_name.py")')` from the JIDE or HIPE command line. Print statements are redirected to the console and can be used to monitor the state of the execution.

Statements passed as parameters to the function are evaluated in the global namespace therefore the following example is legal:

```
IA>> a = 5
IA>> bg('execfile("print a")')
IA>> bg('execfile("a = anExtensiveComputation(12)")')
IA>> bg('execfile("b = aComputation(a)")')
```

There is no guarantee however that the last statement will be executed after the preceding returns the value and that uncertainty can easily lead to cases where "aComputation" is run passing the value 5 (the first assigned to a) or the value returned by "anExtensiveComputation(12)". This is unpredictable and should be carefully avoided.

B.12. Running scripts from a shell command line

it is possible to run user-created DP scripts from the command line of a shell window using the `jylaunch` command.

The following line at the command prompt can be run from a shell.

```
> jylaunch myscript.py
```

where, of course, myscript.py should be replaced with the filename of the script you want to run.

The `jylaunch` command can also be run from the Start menu for the 'hcss' provided by the HCSS installer script.

With the use of the HCSS installer, the `jylaunch` capability is also available under the Program Files start menu as a stand-alone task.

B.13. Errors and exceptions in DP

Here we explain how errors are generated within DP and how these are reported back to the user. Following from this the user should be able to:

• understand error messages that might show up (i) while running an application, or (ii) during a DP session.

• report the error to the custodian of a HCSS module in case a badly described exception occurred, i.e., one which cannot be handled by the user.

B.13.1. Overview of the libraries used in a DP session

The base routines for DP are written in Java, but DP user development uses the more friendly Jython. Typical user development is expected to take place in the console panel with plots and images appearing in separate windows. Within a DP session one can run commands from the JIDE tool that enables the execution of DP/Jython commands, saves and loads scripts, and provides command history support. This tool often provides the core of a user's DP session.
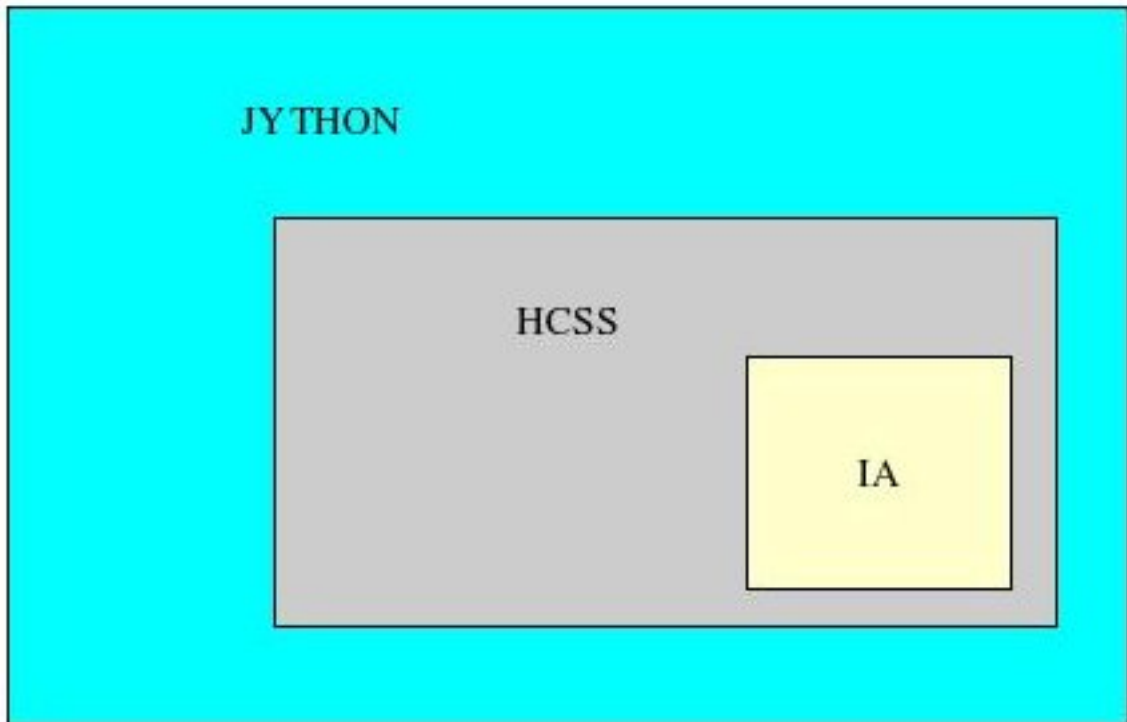
Figure B.11. The overall library structure for a DP session

Library usage for a DP session is illustrated in Figure B.11. Errors, as thrown by Jython and/or JAVA classes, have the same means by which they follow the error back down the program layers to find the root of the error -- "traceback mechanism" (although they differ in the way they present error messages to the user, as shown in the next section).

Interpretation of these error messages allows the user to identify the place where the exception/error originated from.

B.13.2. The error traceback mechanism

In this section we describe the differences in the way Jython and JAVA libraries present error messages.

B.13.2.1. The way Jython presents error messages

Errors in the use of Jython are typically returned directly to the user after their attempted implementation. An example of how Jython presents error messages is given in the following short code example:

```
array = [1,2,3,4,5]
print array[5]
# IndexError: index out of range: 5
```

Another typical Jython error form is a syntax error. Consider the following lines of code

```
x = 2
y = 3
a = x + 2y
```

An error message using this piece of code has the form

```
# Traceback (innermost last):
# (no code object) at line 0
```

```
# SyntaxError: ('invalid syntax', ('<string>', 1, 10, -'a = x + 2y'))
```

which indicates the fault happening in line 1 of the block of code (we only have one line in this case) at the position of character number 10. Note that this information appears in the bottom right panel, by double clicking on the red line corresponding to the error and selecting the `Traceback` entry.

B.13.2.2. The way Java presents error messages

Most DP packages use JAVA classes. If JAVA classes are run within a DP session and an error occurs, an exception is thrown which is propagated upwards to the DP level. An example:

```
dbl = Double("wrong arg")
# java.lang.NumberFormatException: For input string: -"wrong arg"
```

In the history window the command line will be indicated by a red cross, showing that there is an error for this command. Information on the command can be obtained by clisking on the indicator to the left of the red cross. This provides access to the error message and traceback of the error (again, via a mouse click on the indicator).

A Log window can be obtained by using a right-click of the mouse on the history line, in JIDE ONLY, and using the pull-down menu. This provides a separate window with all the information on the problem command.

```
INFO:
<COMMAND>
    <STATEMENT>
        dbl = Double("wrong arg")
    </STATEMENT>

    <EXCEPTION>
        <MESSAGE>
            java.lang.NumberFormatException: For input string: -"wrong arg"
        </MESSAGE>
        <STACK_TRACE>
            Traceback (innermost last):
              File -"<string>", line 1, in -?
            java.lang.NumberFormatException: For input string: -"wrong arg"
            java.lang.NumberFormatException: For input string: -"wrong arg"
                at java.lang.NumberFormatException.forInputString\
    (NumberFormatException.java:48)
                at java.lang.FloatingDecimal.readJavaFormatString\
    (FloatingDecimal.java:1207)
                at java.lang.Double.valueOf(Double.java:202)
                at java.lang.Double.<init>(Double.java:277)
                at sun.reflect.NativeConstructorAccessorImpl.newInstance0\
    (Native Method)
                at sun.reflect.NativeConstructorAccessorImpl.newInstance\
    (NativeConstructorAccessorImpl.java:39)
                at sun.reflect.DelegatingConstructorAccessorImpl.newInstance
    (DelegatingConstructorAccessorImpl.java:27)\
                at java.lang.reflect.Constructor.newInstance
    (Constructor.java:274)\
                at org.python.core.PyReflectedConstructor.__call__\
    (PyReflectedConstructor.java)
                at org.python.core.PyJavaInstance.__init__(PyJavaInstance.java)
                at org.python.core.PyJavaClass.__call__(PyJavaClass.java)
                at org.python.core.PyObject.__call__(PyObject.java)
                at org.python.pycode._pyx113.f$0(<string>:1)
                at org.python.pycode._pyx113.call_function(<string>)
                at org.python.core.PyTableCode.call(PyTableCode.java)
                at org.python.core.PyCode.call(PyCode.java)
                at org.python.core.Py.runCode(Py.java:1136)
                at org.python.core.Py.exec(Py.java:1158)
                at org.python.util.PythonInterpreter.exec(PythonInterpreter.java)
        </STACK_TRACE>
    </EXCEPTION>
</COMMAND>
```

The places in JAVA classes where the code breaks down are indicated. Typically, the traceback starts with the line number of the original program where the problem occurs and follows this with the line numbers in the classes accessed where the problem propagates from. In the above example we have simply tried to attach a string, "wrong arg", to a numeric double. So it is of the wrong format -- as indicated in the first line of the traceback. On other occasions, a more fundamental JAVA error may be occurring deeper in the system. The traceback allows the user to find where this may be happening.

B.13.3. The HCSS exception and logging mechanism

Next to the standard JAVA exception handling mechanism the HCSS is using, it also has a logging mechanism which forwards information, error and warning messages to the user.

B.13.3.1. Exceptions thrown from HCSS classes

In case an error occurs inside the HCSS, for example due to a missing or incorrectly defined configuration variable, the information as part of the exception thrown should explain to the user the cause of the exception. In this way the user should be capable to adjust his/her input arguments and/ or property settings. Property settings can be set using the Property Generation tool ("propgen") -- see Chapter 1. For example:

Let us assume the user has set the configuration variable "var.database.devel" to a database name that does not exist:

```
var.database.devel = -"idonotexist@iccdb.sron.rug.nl"
```

when trying to access this database in a DP-session by:

```
from herschel.access import *
tm = PacketAccess(1030)
packets = HcssConnection.get(tm)
```

Here, a query is done on the database as set by the above property and the exception, appearing in the command line window, reads:

```
# herschel.access.LocationException: Exception in constructor of
      herschel.access.db.LocalConnection:
# herschel.access.LocationException: Failed to get store
# herschel.store.api.StoreException: Failed to create store for
      idonotexist@iccdb.sron.rug.nl:
# herschel.store.api.StoreException: Failed to create
      ObjectStore -"idonotexist@iccdb.sron.rug.nl
# Cannot open database: idonotexist@iccdb.sron.rug.nl
# Error while accessing database: idonotexist@iccdb.sron.rug.nl
# { VException(7001:UT_DB_NOT_FOUND:  DB directory not found) -}
```

In cases where the information as passed by the `Exception` thrown is not sufficient (for example a `NullPointerException` without any textual explanation), then there is a problem with the current system and the user is encouraged to provide feedback to the HSC regarding the lack of exception handling information (currently, this is best achieved through the SPR/SCR system).

In the above example the "`access`" package might improve its exception notification by adding information to the `LocationException`, including a hint for the user that the database is not existing and that the user should check whether `var.database.devel` is properly defined.

B.13.3.2. The HCSS logging mechanism

The logging mechanism allows (HCSS) classes to pass errors, warnings and/or info to the end-user. To view the error logging mechanism, go to the Help menu or click on the ![icon] icon (see also Section 2.2.5).

For the HCSS end-user this mechanism, rather than the analysis of exception handling, is likely to be used more often, especially when HCSS software is fully matured. The difference between the two is that exception handling is more often used by the developer for debugging purposes, whereas the logging mechanism is intended to be used by the end-user to get insight into the behaviour of an (HCSS) application or class. The logging mechanism enables the developer to include messages when an exception is thrown on how the class internally handles possibly thrown exceptions.

To give an example why, next to the exception mechanism, the logging mechanism was introduced: suppose we have a layered HCSS component (i.e. within an instance of a class there are calls to instances of other classes and these will call others on their turn), deep within this component an exception occurs and at a higher level this exception is caught again. In such a scenario the end-user of the component will not be aware of the fact that this exception occurred. However, by use of the logging mechanism the developer of the component can pass a message (an error, warning or info; depending on how severe this exception was) next to the exception thrown, as well as being able to pass relevant information to the user when the exception is caught.

Appendix C. Jython operators

The following tables shows all the various operators you can use in Jython. For completeness we have also listed one operator introduced in version 2.2 of Jython, but absent from version 2.1, the one used by the HCSS software.

This list and the associated operator descriptions have been largely taken from the Python Reference Manual, which you can find online at http://docs.python.org/ref/.

Table C.1. Jython unary arithmetic operators

Operator	Operator description	Example
+	**Unary plus**: yields its numeric argument unchanged.	`print +5` `# 5`
–	**Unary minus**: yields the negation of its numeric argument.	`print -5` `# -5`
~	**Invert**: yields the bitwise invert of its plain or long integer argument.	`print ~5` `# -6`

Table C.2. Jython binary arithmetic operators

Operator	Operator description	Example
+	**Sum**: yields the sum of its arguments.	`print 2 + 2` `# 4`
–	**Subtraction**: yields the difference of its arguments.	`print 2 - 3` `# -1`
*	**Multiplication**: yields the product of is arguments.	`print 3 * 2` `# 6`

/	**Division**: yields the quotient of its arguments.	`print 5 / 2`
		`# 2`
		`print 5.0 / 2`
		`# 2.5`

//	**Floor division (Jython 2.2 alpha only)**: yields the result of the `floor()` function applied to the quotient of its arguments.	`print 5 // 2`
		`# 2`
		`print 5.0 // 2`
		`# 2.0`

%	**Modulo**: yields the remainder from the division of its arguments.	`print 5 % 2` `# 1`
**	**Power**: yields its left argument raised to the power of its right argument.	`print 5**2` `# 25`

Table C.3. Jython shifting operators

Operator	Operator description	Example
<<	**Left shift**: `a << b` shifts plain or long integer `a` by `b` bits.	`print 5 << 1` `# 10`
>>	**Right shift**: `a >> b` shifts plain or long integer `a` by `b` bits.	`print 5 >> 1` `# 2`

Table C.4. Jython binary bitwise operators

Operator	Operator description	Example
&	**Bitwise AND**: yields the bitwise AND of its plain or long integer arguments.	`print 5 & 6` `# 4`
^	**Bitwise XOR**: yields the bitwise exclusive OR of its plain or long integer arguments.	`print 5 ^ 6` `# 3`
|	**Bitwise OR**: yields the bitwise inclusive OR of its plain or long integer arguments.	`print 5 | 6` `# 7`

Table C.5. Jython comparison operators

Operator	Operator description	Example
<	**Less than**: a < b yields true if a is less than b.	`print 5 < 6` `# 1`
>	**Greater than**: a > b yields true if a is greater than b.	`print 5 > 6` `# 0`
==	**Equal to**: a == b yields true if a and b are equal.	`print 5 == 6` `# 0`
>=	**Greater or equal to**: a >= b yields true if a is greater than or equal to b.	`print 5 >= 6` `# 0`
<=	**Less or equal to**: a <= b yields true if a is less than or equal to b.	`print 5 <= 6` `# 1`

!= (preferred) or <>	**Not equal to**: a != b yields true if a is not equal to b.	`print 5 != 6`
		`# 1`
		`print 5 <> 5`
		`# 0`

Table C.6. Jython boolean operators

Operator	Operator description	Example
and	**Boolean AND**: yields True if both arguments are true, False otherwise.	`print 1 and 0` `# 0`
or	**Boolean OR**: yields True if at least one argument is true, False otherwise.	`print 1 or 0` `# 1`
not	**Boolean NOT**: yields True if the argument is false, False otherwise.	`print not 1` `# 0`

Appendix D. Naming Conventions

for Java and Jython users and developers. Version 0.3, 6th December 2006

Element	Description	Naming convention

Class
 	
Section 1.13.1
	Defines the state and behaviour of something. Classes are defined as declaring variables (fields) and functions (methods) associated with the objects of that class.	Names should be nouns and written in mixed case starting with an upper case letter. Do not use underscores to separate words.
		
		`DataFrameGenerator,`
		`FitsArchive`

Interface
 	
Section 1.13.2.1
	Defines a collection of method definitions and constant values. It can later be implemented by classes that define this interface with the `implements` keyword.	Names have the same convention as class names but are preferably adjectives. Try to end the names with -able or -ible:
		
		`Sortable, Accessible,`
		`Savable`

Variable	An item of data named by an identifier. Each variable has a type, such as `int` or `Frame`, and a scope.	Names should be mixed case starting with a lower case letter. Do not use underscores to separate words.
		
		`frameBufferCounter,`
		`nSamples,       line,`
		`detectorNo`

Instance Variable
 	
Section 1.13.1
	A variable that is part of an object.
	
	For the rationale of this naming convention see HSCDT/TN009 on ESA Livelink
		Names should start with an underscore, otherwise follow the general conventions for variables (see above).
		
		`_packetType,`
		`_isVisible`

Local Variable	A variable that is part of a function or method.	Names follow the naming convention of normal variables.
		
		`counter,        length,`
		`pixelName`

Constant	A variable whose value that can not be changed during execution.	Names should be all uppercase using an underscore to separate words: `MAX_ITERATIONS`

Boolean variable and method	A logical type/function that can only have or return the values 'true' or 'false'. Methods have parentheses () while variable haven't.	Names should start with is-, has-, can-, or should-.
		
		`isVisible,`
		`hasChanged(),`
		`canHandle(),`
		`shouldAbort`

Parameter		

Element	Description	Naming convention
	An argument to a function or a method.	Names follow the naming convention of normal variables. `name, packet`
Property	A platform independent implementation of environment variables and settings.	Names should be all lower case and start with 'hcss'. The hierarchical parts should be separated with a dot. `hcss.binstruct.services`
Method Section 1.13.1	A function defined in a class.	Names should be verbs and written in mixed case starting with a lower case letter. Do not use underscores to separate words. `getName(), load()`
Function Section 1.11	A jython function is a collection of code lines to perform a specific task under one name. Functions take arguments and provide one output. They are like methods, except they are not inside a class. A function can also be an instance of the Task class.	Names follow the same convention as method names in classes. `resample(), readTm()`
Numeric function Section 3.4	Parameterless Java functions provided by the herschel.ia.numeric toolboxes. For these function only one instance is needed. Other numeric functions follow the same convention as classes.	Names are in all uppercase with an underscore to separate words. `UNIQ,      MEDIAN, IS_FINITE`
Task Chapter 4	A Task is a class which can be called as a function. Tasks do input and output parameter type checking and provide history to Products.	Names follow the same conventions as for classes. Task names should end with the word 'Task'. `DisplayDataFrameTask, ResampleTask`
Package Section 1.13.4	Defines a collection of related classes and interfaces in Java. Packages provide the namespace in Java and Jython.	Names should be in lower-case letters and digits, don't use underscores. `herschel.ia.numeric` Package names should be short so that the fully qualified package name doesn't become excessively long.

Abbreviations and acronyms should **not** be all uppercase when used as a name:

GOOD	BAD
exportAsHtml()	exportAsHTML()
saveAsJpeg()	saveAsJPEG()
OolPacket	OOLPacket

Using all uppercase for the abbreviations in base names will give conflicts with the naming conventions given above. A variable of this type would have to be named `hTML`, `jPEG` etc. which obviously is not very readable. Another problem is illustrated in the examples above: when the name is connected to another, the readability is seriously reduced, since the word following the acronym does not stand out as it should.

The term *compute* can be used in methods where something is computed and might take considerable time to execute.

```
computeAverage(), matrix.computeInverse()
```

Give the reader the immediate clue that this is a potential time consuming operation, and if used repeatedly, he might consider caching the result. Consistent use of the term enhances readability.

The *'n'* prefix should be used for variables representing a number of objects, note that the names are plural.

```
nPoints, nLines, nSamples
```

The notation is taken from mathematics where it is an established convention for indicating a number of objects. Note that Sun uses the `num` prefix in the core Java packages for such variables. This is probably meant as an abbreviation of number of, but as it looks more like number it makes the variable name strange and misleading. If "number of" is the preferred phrase, `numberOf` prefix can be used instead of just `n`. The `num` prefix must not be used.

The *'No'* suffix should be used for variables representing an entity number.

```
tableNo, employeeNo
```

The notation is taken from mathematics where it is an established convention for indicating an entity number.

Reserved words: the following words are reserved by Java as language keywords and can not be used for variables, methods or class names in Java.

```
abstract, continue, for, new, synchronized, assert, default, goto,
package, this, boolean, double, if, private, throws, break, do,
implements, protected, throw, byte, else, import, public, transient,
case, enum, instanceof, return, try, catch, extends, interface,
short, void, char, finally, int, static, volatile, class, final,
long, super, while, const, float, native, switch.
```

Java code example

```
package herschel.ia.numeric; -// herschel.ia.numeric: PACKAGE
public final class Complex1d -// Complex1d: CLASS
    implements Serializable -// Serializable: INTERFACE
{
    private transient double[][] _internal; -// _internal: INSTANCE VARIABLE
    -// writeObject: METHOD
    private void writeObject(ObjectOutputStream os) { -// os = METHOD PARAMETER
        os.defaultWriteObject();
        os.writeInt(length());
        if (length()==0) return;
```

```
        for (int i=0,n=length();i<n;i++) { -// i = LOCAL VARIABLE
            os.writeDouble(_re[i]); os.writeDouble(_im[i]);
        -}
    -}
}
```

Jython code example

```
# herschel.ia.dataset.gui = PACKAGE; DatasetInspector = CLASS
from herschel.ia.dataset.gui import DatasetInspector
# PI = CONSTANT
from java.lang.Math import PI
# testName = VARIABLE
testName = -"chop_freq_test_2909_1832_1902_"
# load = METHOD
t2 = fits.load(myDir+testname+"PHOTF.fits").default
# MAX = NUMERIC FUNCTION
maxStep = MAX(step[step.where(step < 0xffff)])
# startEndTimes = FUNCTION; step, maxStep, time... = FUNCTION PARAMETERS
def startEndTimes(step, maxStep, time, startTime, endTime):
  for i in range(0, maxStep): # i = LOCAL VARIABLE
    temp=(step.where(step == i+1))
    endTime[i] = time[MAX(temp.toInt1d())
  return endTime
# len = FUNCTION
upper = len(startarr)
```