

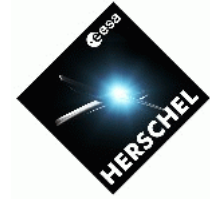


# Scripting in HIPE

Miguel Sánchez-Portal

HSC, ESAC

# Introduction



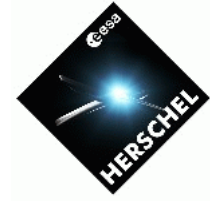
- HIPE Scripting is based on Jython
  - [Jython](#) is a Java implementation of the [Python](#) language.
  - Partial implementation of Jython 2.1 (Python is currently in v3.x)
  - For more information on the Python language, please visit <http://www.python.org/doc/>
- Only a very basic knowledge is required. For instance:
  - You don't need to write object-oriented code
  - Many elements are specific to HIPE so advanced Python features aren't needed
- Python resources in the HIPE documentation contain most of what is needed
  - See the Scripting and Data Mining document (HERSCHEL-HSC-DOC-0517) in the Help system.
- *Acknowledgements: this presentation is largely based on material kindly provided by D. Shupe (NHSC)*

# Outline



- Some Python basics (core language features, usable in Jython or C-based Python)
  - Variables and other basics (e.g. basic I/O)
  - Lists
  - Blocks, Loops, indexing
- Data structures/objects hierarchy (simple to complex)
  - Numeric arrays and methods
  - TableDatasets
- A (useful) example

# Basic Python concepts – Variables



- No 'data-typing' or declaration needed
- Assignment:

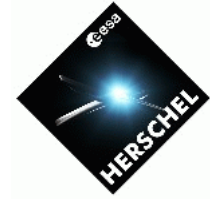
```
a = 1 #this is a comment
# Check the variable class (in this case a PyInteger)
print a.__class__
b = 2.0 # now a PyFloat
```
- Strings can use single or double quotes:

```
c = 'hi there'
e = "I can use 'single quotes' here"
```

Boolean:

```
d = Boolean(0) # Java boolean
```

# Numeric variable types



- Jython types:
  - *Integer*:  $a = 3$
  - *Long integer*, denoted by the l or L suffix:  $a = 3L$
  - *Float*:  $a = 3.0$
  - *Complex*:  $a = (3 + 1j)$
- Java numeric types available to Jython:
  - *Byte*: signed 8-bit integer  $\rightarrow a = \text{Byte}(0)$ .
  - *Short*: signed 16-bit integer  $\rightarrow a = \text{Short}(0)$ .
  - *Integer*: signed 32-bit integer  $\rightarrow a = \text{Integer}(0)$ .
  - *Long*: signed 64-bit integer  $\rightarrow a = \text{Long}(0)$ .
  - *Float*: single-precision 32-bit floating point  $\rightarrow a = \text{Float}(0)$ .
  - *Double*: double-precision 64-bit floating point  $\rightarrow a = \text{Double}(0)$ .
  - *Boolean*: either true or false  $\rightarrow a = \text{Boolean}(0)$  (or  $a = \text{Boolean}(\text{False})$ )
- Caveat: sometimes, Jython and Java types don't mix well. Check Section 1.4. *Type conversions* of the *Scripting manual* for a description

# More Python basics



- The comment character is the hash sign

```
# this is a comment
```

- The continuation character is the backslash

```
x = a + b + \  
    c * d * e
```

- A formatted string uses C-style format characters and the percent sign

```
print "integer = %d, real = %f" %(j,x)
```

- Print to an ascii file

```
fh = open('myoutput.txt', 'w')  
print >> fh, "integer = %d," %j  
fh.close()
```

# More I/O, and a first example



```
import os #import the Python module "os"

# Open the file
myFile=open("/Users/msanchez/hcss_work/funnytest.txt", "w")

# Print class
print myFile.__class__

# Print available methods
print dir(myFile.__class__)

# Write two lines to the file. Observe the LF character at the end
# of the string
line="%07.2f %07.2f %07.2f %07.2f\n" % (902.42123456, 1044, 920.90, 1043.96)
myFile.write(line)
line="%07.2f %07.2f %07.2f %07.2f\n" % (9999.01, 1050.29, 30.90, 666.66)
myFile.write(line)

# Don't forget to close the file!!
myFile.close()

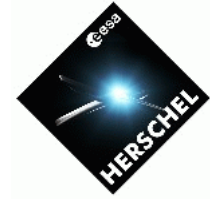
#This is an example of invoking a "shell" command from your HIPE script...
os.system("cat /Users/msanchez/hcss_work/funnytest.txt")

# Reading the file "HIPE style"
ascii=AsciiTableTool()
ascii.parser=CsvParser(skip=0,delimiter=' ')
ascii.template=TableTemplate(4, \
names=["val1", "val2", "val3", "val4"], \
types=["Double", "Double", "Double", "Double"], \
units=["pixel", "pixel", "pixel","pixel"])
myTable=ascii.load("/Users/msanchez/hcss_work/funnytest.txt")

# But, what's myTable?
print myTable.__class__

# We'll see more about datasets, and specifically about table datasets
# Just as an example, retrieving the values in the first column of our file
print myTable.getColumn("val1").getData()
```

# Lists



- Lists are very general and powerful structures
- Easy to define, and the members can be anything:  
`x = [1, 2, 'dog', "cat"]`
- Appending or removing items is easy:  
`x.append(5)`  
`x.remove('dog')`
- Empty list  
`z = []`
- “Java-style” lists: ArrayList: also flexible to store whatever you want:

```
myList=ArrayList()  
myList.add(1.0)  
myList.add(2.0)  
myList.remove(1.0)
```



# Conditional Blocks



- Syntax:

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

- Notice that blocks are denoted by indentation only
- Example in SPIRE large map pipeline scripts:

```
if pdtTrail != None and \  
    pdtTrail.sampleTime[0] > pdt.sampleTime[-1]+3.0:  
    pdtTrail=None  
    nhktTrail=None
```

# For Loops



- Syntax of a `for` loop:  
`for var in sequence:  
 block`
- The *sequence* can be any list, array, etc.  
Example from pipeline scripts:  
`for bbid in bbids:  
 block=level0_5.get(bbid)  
 print "processing BBID="+hex(bbid)`
- The `range` function returns a list of integers. In general `range(start, end, stepsize)` where *start* defaults to 0 and *stepsize* to 1.  
`print range(5)  
# [0, 1, 2, 3, 4]`
- The `range` function can be used to loop for an index:  
`for i in range(20):`

# while, break and continue



- while loop syntax:  
**while condition:**  
**block**
- The break statement breaks out of the smallest enclosing for or while loop.
- The continue statement continues with the next iteration of the loop.
- The pass statement does nothing. It can be used when a statement is required syntactically but the program requires no action:

```
results=browseProduct(ps)  
while results.size()==0:  
    pass
```



# Indexing and Slicing

- Any *sequence* (list, string, array, etc.) can be indexed
  - zero is the first element
  - negative indices count backwards from the end

```
x=range(4) # [0, 1, 2, 3]
print x[0] # 0
print x[-1] # 3
```

- A slice consists of  $[start:end:stride]$  in general.  
*Start* defaults to 0, *end* to last, *stride* to 1. Examples:

```
print ss[:2] # ['a', 'b']
print ss[::2] # ['a', 'c']
print ss[::-1] # ['d', 'c', 'b', 'a']
```

# Functions



- Functions are defined by *def* statement plus an indented code block:

```
def square(x):  
    result=x*x  
    return(result)
```

- Optional arguments are given default values in the definition:

```
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
```

- Arguments are passed by value – the names in the *def* statement are local to the body of the function

# Import statements



- **import** makes Python modules or Java packages available to your session or script
- First form uses full names:  

```
import herschel.calsdb.util  
print herschel.calsdb.util.Coordinate
```
- Second form puts name in your session  

```
from herschel.calsdb.util import Coordinate
```
- Third form includes all (use with caution!)  

```
from herschel.calsdb.util import *
```

# Many imports are done for you



- HIPE imports many packages on startup
- “jylaunch” (for batch mode) does too
- When writing modules or plugins,  
explicitly import everything you need
- No cost for importing a module that was  
imported previously

# Hierarchy of data structures (partial list)

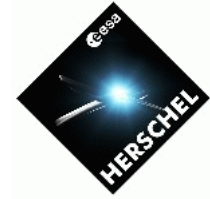


- Numeric arrays
- *Array Datasets*
- TableDatasets
- Products (e.g. DetectorTimeline)
- *Context Products*

The items lower on this list, are containers of the items one level above



# Numeric arrays



- In the `herschel.ia.numeric` package
- Separate classes for data type and dimension
  - `Float1d`, `Float2d`...`Double1d`, `Double2d`...`Int1d`,  
`Int2d`...,`Long1d`, `Long2d`...`Bool1d`, `Bool2d`....etc
- Several ways to initialize:

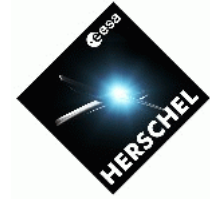
```
z = Double1d(10) # [0.0, ..., 0.0]
```

```
z = Double1d.range(10)#[0.0,1.0,...9.0]
```

```
z = Double1d([1,2,3]) # list
```

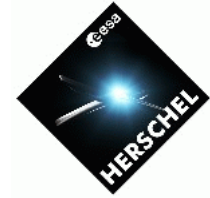
```
z = Double1d(range(10,20))
```

# Numeric functions



- Basic functions are in `herschel.ia.numeric.toolbox.basic`
  - double->double array-to-array functions:  
**ABS, ARCCOS, ARCSIN, ARCTAN, CEIL, COS, EXP, FLOOR, LOG, LOG10, SIN, SORT, SQRT, SQUARE, TAN**
  - Array functions returning a single value  
**MIN, MAX, MEAN, MEDIAN, SUM, STDDEV**
- Advanced functions for filtering, interpolation, convolution, fitting, etc. in other `herschel.ia.numeric.toolbox` packages

# Numeric arrays cont'd

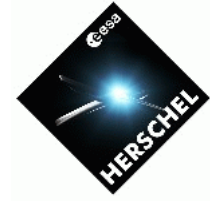


- For 1d, slicing/indexing is the same as Python lists
- For 2d+ arrays, dimensions are set off by commas
  - E.g. array3d[k,j,i]
  - The “fastest” index is the last
    - Same ordering as C, C++, Java, other languages
    - opposite ordering as Fortran, IDL

```
whatsThat=[[1,2,3],[4,5,6],[7,8,9]]
#
print whatsThat.__class__
org.python.core.PyList

myArray=Double2d(whatsThat)
#
print myArray.__class__
herschel.ia.numeric.Double2d
#
print myArray
[
[1.0,2.0,3.0],
[4.0,5.0,6.0],
[7.0,8.0,9.0]
]
# Print "column 0" of the array
print myArray[:,0]
[1.0,4.0,7.0]
# Print "Line 1" of the array
print myArray[1,:]
[4.0,5.0,6.0]
```

# Datasets



- Dataset attributes: description+ data (arrays)+metadata

- Array Data set:

```
x = DoubleId.range(10)
```

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

- TableDatasets gather Numeric arrays with units

```
x = DoubleId.range(100)
```

```
tbl = TableDataset(description="test table")
```

```
tbl["x"]=Column(data=x,\  
    unit=herschel.share.unit.Duration.SECONDS)
```

```
tbl["sin"] = Column(data=SIN(x))
```

- Access

```
print tbl["x"].unit
```

```
print tbl["x"].data[4] #5th element of data
```

- Easily visualized with TablePlotter

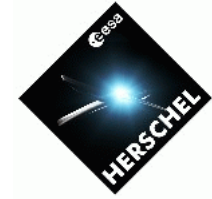
# Products



- Products are the containers of Datasets
- Every Product has a 1-to-1 correspondence to a FITS file (but there are caveats on usability)
- Datasets are added and referenced by name:

```
prod = Product()  
prod["signal"] = tbl  
print prod["signal"]["x"].unit  
p=PlotXY(pdt['voltage']['sampleTime'].data, \  
         pdt['voltage']['PSWE4'].data)
```

# Learn from the GUI!



- Many Views and Tasks execute commands in the Console
  - Copy and paste into scripts when useful
- After a compound object in a viewer, copy and paste the expression that accesses the piece you want

