DP Scripting

David Shupe

NHSC

Introduction

- DP Scripting is based on Python
  - Jython, the Java equivalent of C-based Python
  - HCSS/HIPE includes an implementation of Jython 2.5

- Only a few language features needed to get going
  - Java is not required – use scripting to glue together the provided Java modules from the pipelines, PlotXY, or the Numerics library
  - *It is fine to write "quick-and-dirty", procedural code in Python. Object-oriented code is not required*
  - 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 Manual in HIPE help

Sorry astronomers, but IDL isn't going to cut it if you want to get a tech job. You need to learn one of the industry-standard programming languages. Python, Ruby, Java, Perl, and C++ are all good languages to pick-up…

-Jessica Kirkpatrick,

astronomer-turned-data scientist

Outline

- Selected Python features (core language features, usable in Jython or C-based Python)
  - Lists and indexing
  - Tuples and dictionaries
  - Import statements
- Data structures/objects hierarchy (simple to complex)
  - Numeric arrays and methods
  - TableDatasets
- Common pitfalls
  - Assignment of array variables – not the same as a copy
  - Unintended copies of large objects

Variables

- No 'data-typing' or declaration needed
- Assignment:
  ```
  a = 1
  b = 2
  ```

- Strings can use single or double quotes:
  ```
  c = "hello world"
  e = 'hi there'
  ```

More Python basics

- The comment character is the pound 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()
```

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 = []
  ```

Tuples and Dictionaries

- Tuples are just like lists – except they can't be modified:

```
d = ('one', 'two', 'three')
```

- Dictionaries give names to members

```
wavel = {'PSW':250, 'PMW':350,\
         'PLW':500}
```

- Easy to add members

```
wavel['pacsred'] = 160
print wavel['PSW']
```

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):
```

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:
  ```
  ss = ['a', 'b', 'c', 'd']
  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 (x,y,z)
```

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

Import statements

- **import** makes Jython 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
  ```
  from herschel.calsdb.util import *
  ```

Many imports are done for you

- HIPE imports many packages on startup

- When writing modules or plugins, explicitly import everything you need

- No cost for importing a module that was imported previously

Commands can be run in the background

- Use the bg function with your command inside a string

  ```
  bg('scans=baselineRemovalMedian(obs.level1)')
  ```


- Right-click on a script in Navigator to run in background

  ```
  HIPE> bg('execfile("~/jyscripts/bendoSourceFit_v0_9.py")')
  Started: execfile("~/jyscripts/bendoSourceFit_v0_9.py")
  Finished: execfile("~/jyscripts/bendoSourceFit_v0_9.py")
  ```

Hierarchy of data structures
(partial list)

- Numeric arrays

- *Array Datasets*

- TableDatasets

- Products (e.g. DetectorTimeline)

- *Context Products – not covered here*

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
- Tips to improve performance
  - Avoid looping over array indices
  - Take care not to create too many temporary copies of arrays (more on this later)

TableDatasets

- TableDatasets gather Numeric arrays with units

```
x = Double1d.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 opening up a compound object in a viewer, copy and paste the expression that accesses the piece you want
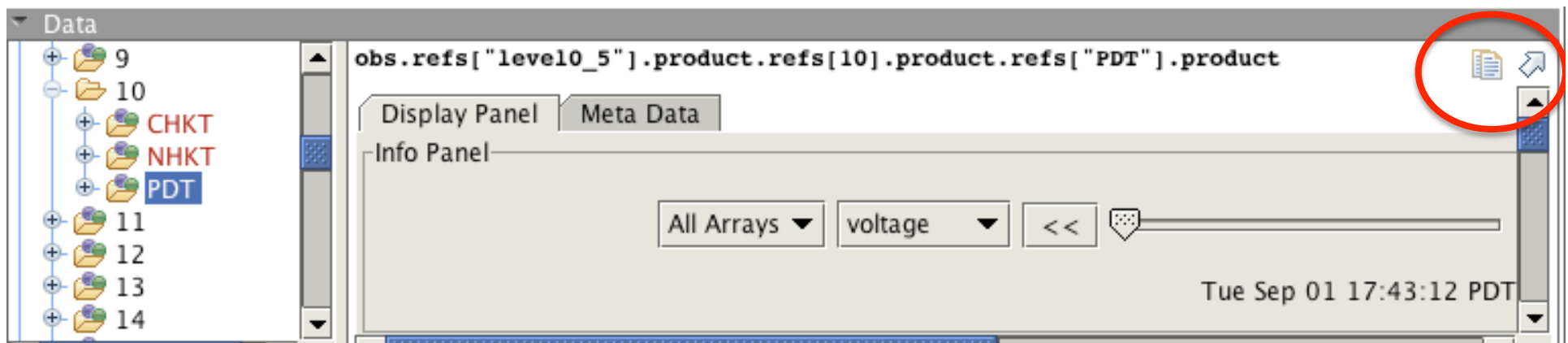
Listing methods with the **dir** function

- The dir function lists the methods specific to a given class
  print dir(*variable*.__class__)

- In HIPE it is reachable from right-click on variable, "Show methods"

Avoiding common pitfalls

# Assignment of array is not a copy

- Simple example:

```
a = Int1d.range(2)
print a
# [0, 1]
b = a
b[0] = 5
print b
# [5, 1]
print a
# [5, 1] ????
```

- What happened? Assignment is "by value". What is the value of **a**? It is an object which is an instance of the Int1d class. Then **b=a** binds the name **b** to the same object to which **a** is bound.

A useful visualization

- Do not think of variables as physical locations in memory

- Variables are *names* that are *bound* to *objects*

- The drawing shows the state after:
  b = a

Names

a　　　**b**

Int1d
[0, 1]

Values/Objects
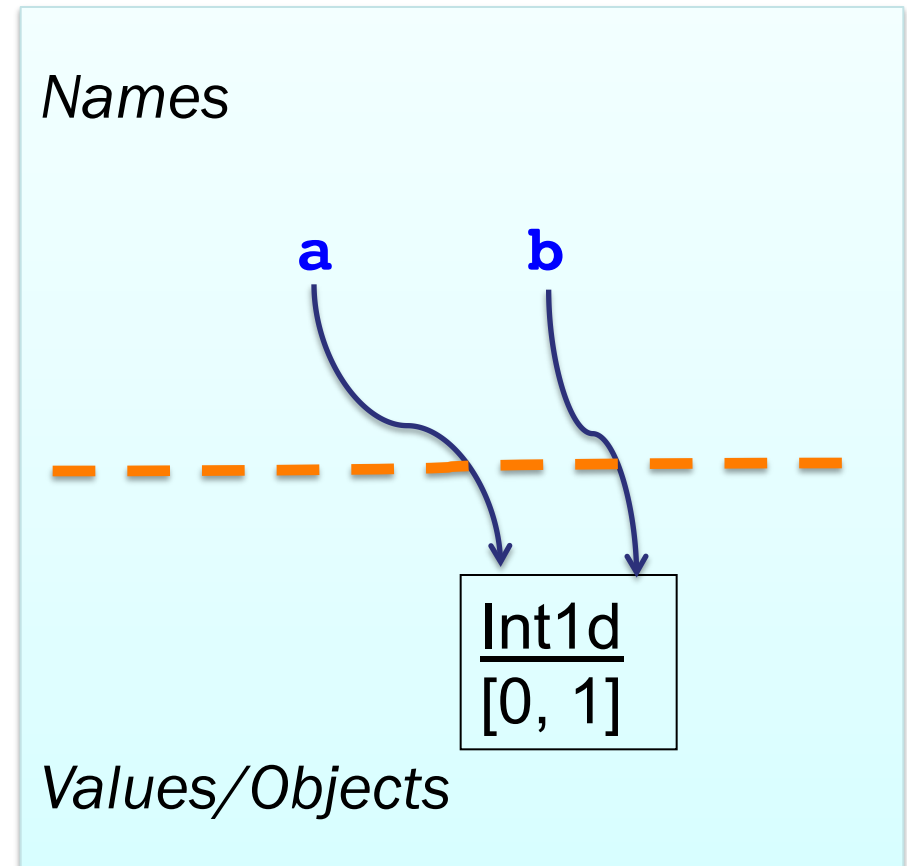
What does **b[0] = 5** really do?

- The line
b[0] = 5
is equivalent to
b.__setitem__(0,5)
which is a *method* of our object, that modifies a single element

- Our two variables are still bound to the same object

Names

a   b

Int1d
[5, 1]

Values/Objects
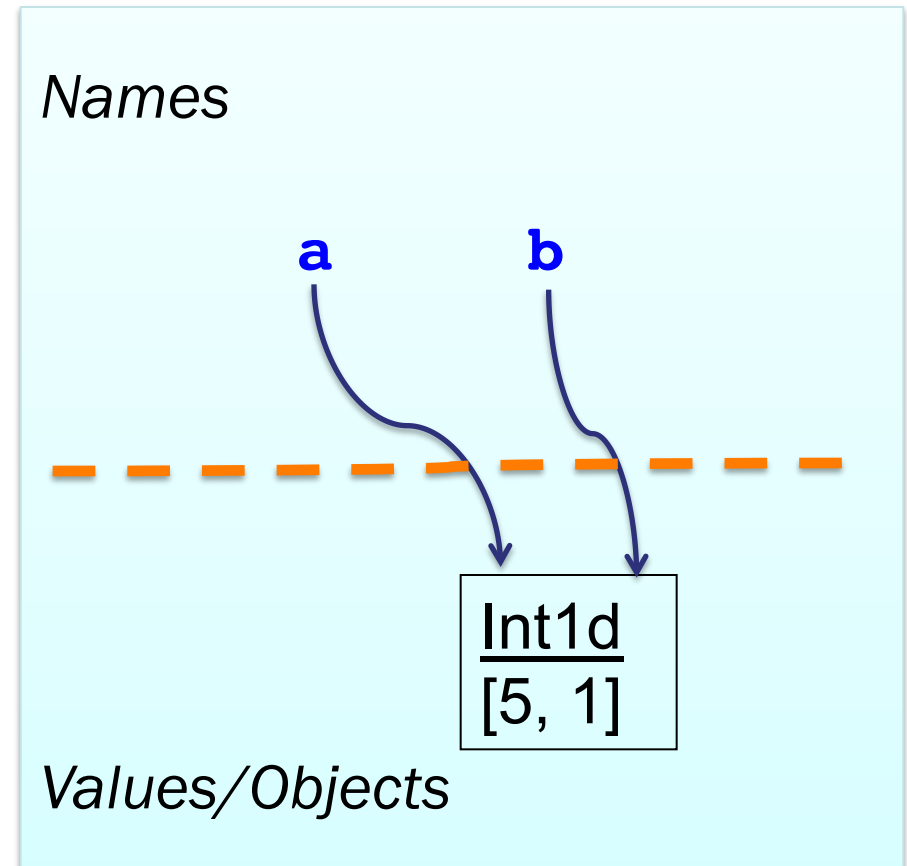
How do I get a new array object?

- For a new copy of the array object, do
  b = a.copy()

- This also works:
  b = Int1d(a)

- The diagram at right shows the state after
  b[0] = 5

Names

a   b

Int1d
[0, 1]

Int1d
[5, 1]

Values/Objects
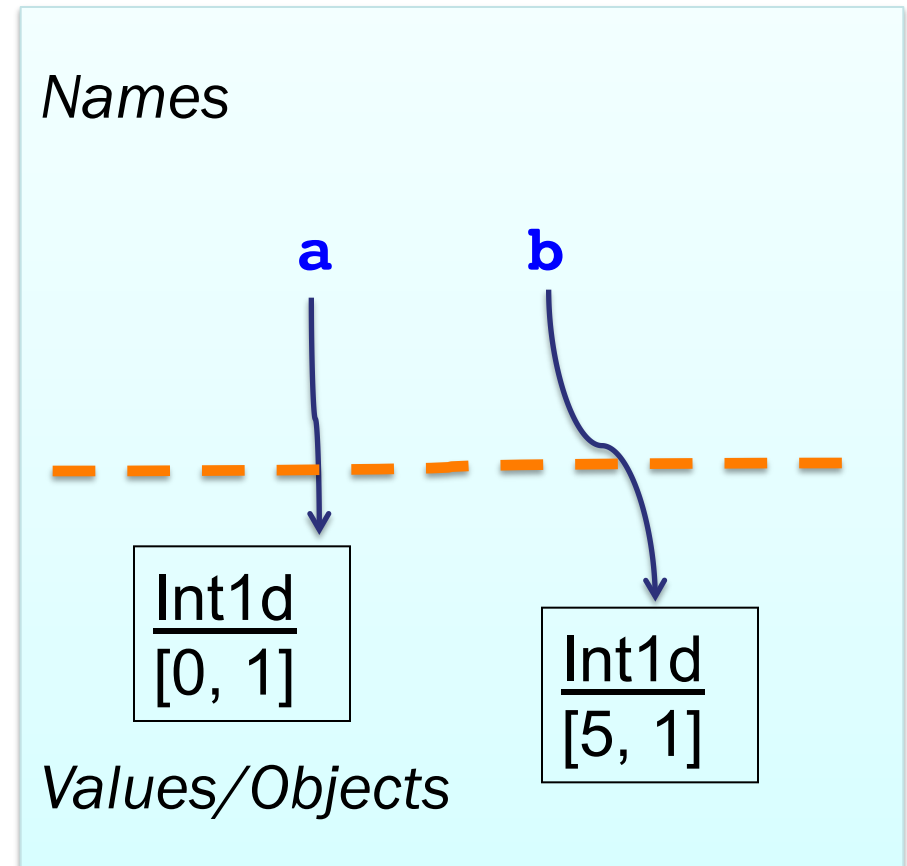
Automatic creation of arrays

- Another example:

```
a = Int1d.range(2)
print a
# [0, 1]
b = a
b = b + 5
print b
# [5, 6]
print a
# [0, 1]
```

- What happened? At

```
b + 5
```

a new array was automatically created to hold the sum of **b** and 5. Then the name **b** was bound to this new array object. **a** was left unchanged.

In-line operations

- A changed example:

```
a = Int1d.range(2)
print a
# [0, 1]
b = a
b += 5
print b
# [5, 6]
print a
# [5, 6]
```

- What happened? At

```
b += 5
```

the in-line operator **+=** means that the operation is done in place – no new copy is made of the object to which **a** and **b** are bound.
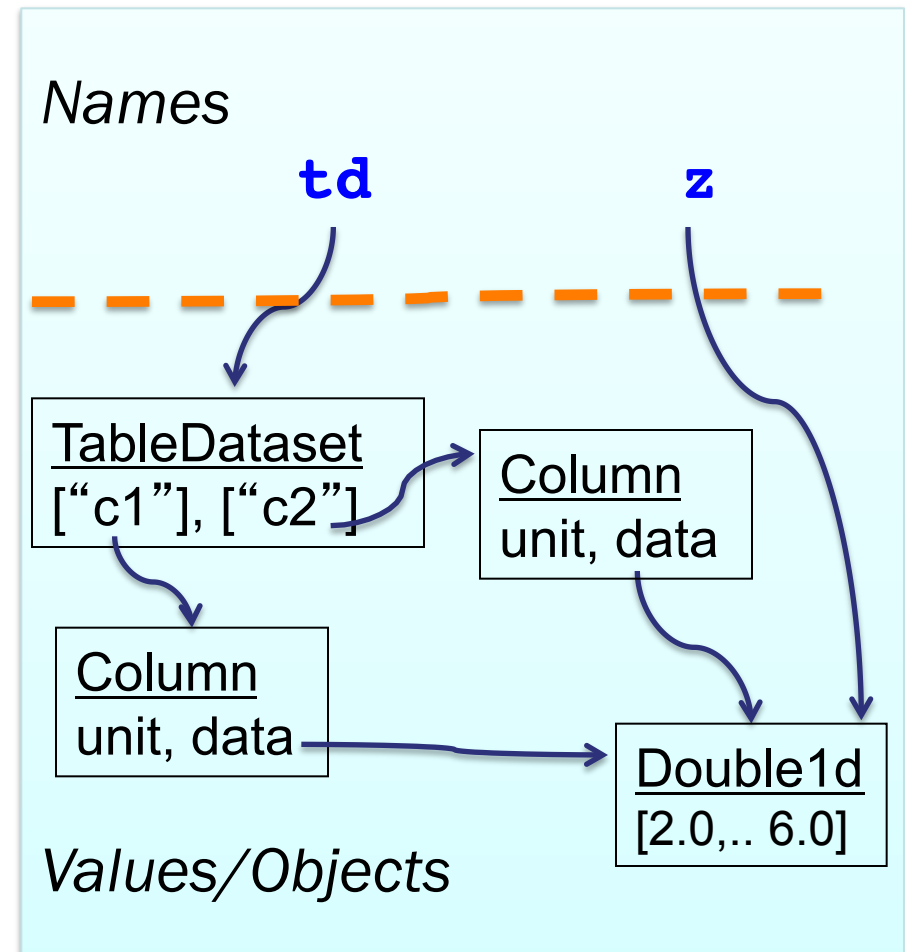
- Saves memory

Garbage collection

- A related example:
  ```
  a = Int1d.range(2)
  b = a.copy()
  b = b + 5
  ```

- For a time, three array objects are taking up memory

- What happens to the first copied array? Eventually the *garbage collector* frees up the memory

Names

a   b

Int1d
[0, 1]

Int1d
[0, 1]

Int1d
[5, 6]

Values/Objects
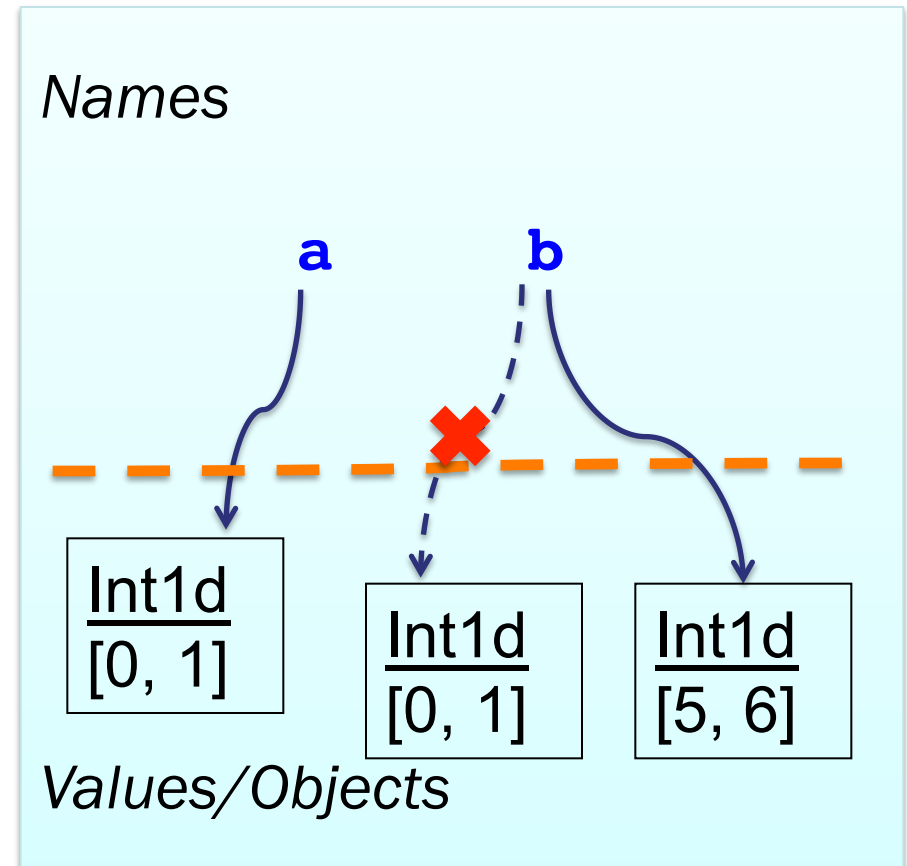
Changes inside higher-level products

- Another example:

```
z=Double1d.range(5)
td=TableDataset()
td["c1"]=\
   Column(data=z)
print td["c1"].data
#[0.0,1.0,2.0,3.0,4.0]
z += 2
td["c2"]=\
   Column(data=z)
print td["c1"].data
#[2.0,3.0,4.0,5.0,6.0]
```


Names

td   **z**

TableDataset
["c1"], ["c2"]

Column
unit, data

Column
unit, data

Double1d
[2.0,.. 6.0]

Values/Objects

Avoiding temporary copies of arrays

- Assume we have three large arrays named
  x, y, c
  and we want to compute
  y = (x + SIN(y))/c

- As typed above, some temporary arrays are made, then discarded

- Can greatly increase memory usage

- Here's a way to do it with in-line operations, making no array copies.
  y.perform(SIN)
  y += x
  y /= c

- The **y.perform** does an in-place operation.
  y.apply(SIN) makes a copy, like **SIN(y)**

Reference slides

Advanced topics....

List comprehensions

- List comprehensions are a shorthand for writing a loop that appends to a list
```
print [x*x for x in range(10)]
# [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

- The above is short for:
```
list = []
for x in range(10):
    list.append(x*x)
print list
```

- Handy for converting any sequence into a list

- For numerical calculations, it is more efficient to use the Numeric functions

Context Products

- Context Products are the containers of Products
  - More precisely, contains references to products
  - Not understandable outside HIPE/HCSS
- Two flavors of Context Product:
  - Map Context – maps keys/names to product refs
    ```
    mc = MapContext()
    mc.refs["prod1"] = ProductRef(prod))
    p = mc.refs["prod1"].product
    ```
  - List Context – ordered list of Products
    ```
    lc = ListContext()
    lc.refs.add(ProductRef(prod))
    p = mc.refs[0].product
    ```

Building up complex products

- Array => TableDataset => Product => Context:

```
x = Double1d.range(100)
table = TableDataset()
table["col1"] = Column(data=x)
prod = Product()
prod["error"] = table
mcontext = MapContext()
mcontext.refs["unc"] = \
        ProductRef(prod))
```