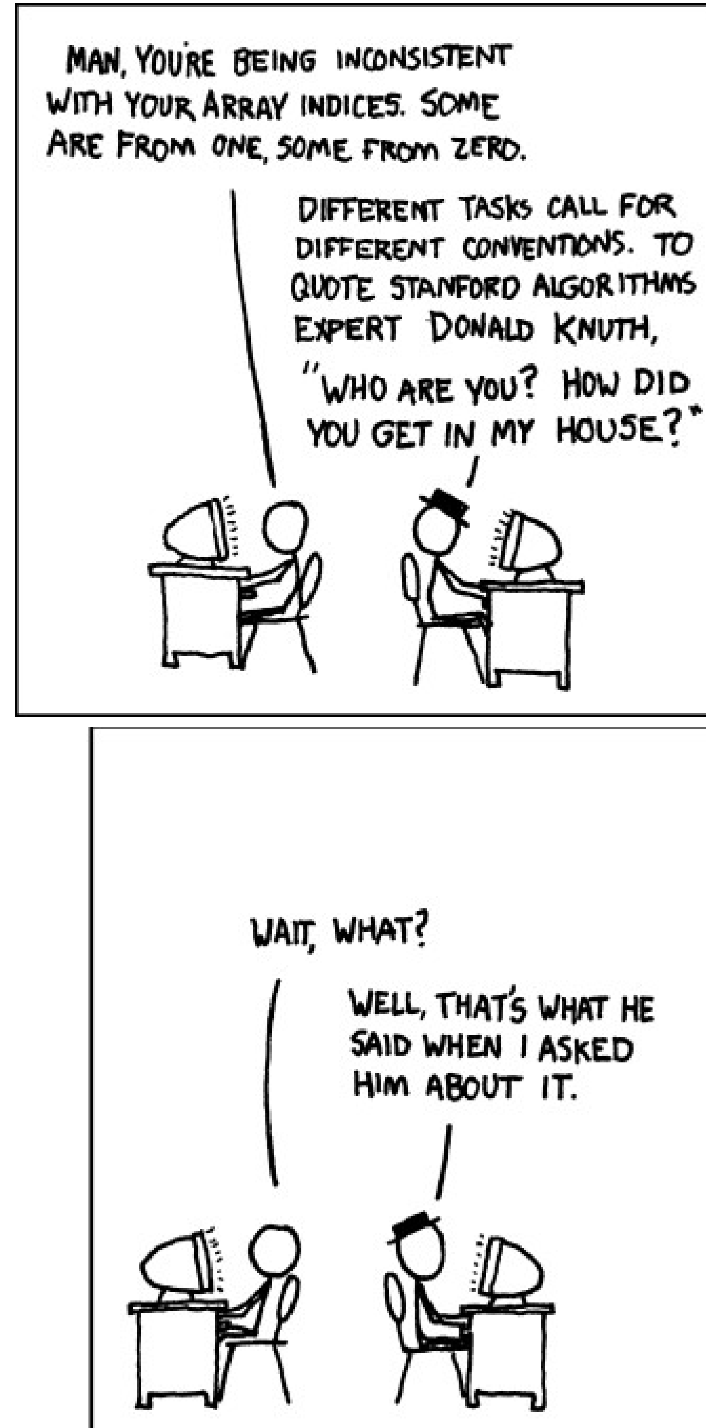
Introduction to IDL

3 – Data structures and strings

Paulo Penteado
pp.penteado@gmail.com
http://www.ppenteado.net



(http://xkcd.com/163)

Containers

A single value in a variable is not enough.

Containers – variables that hold several values (the elements)

There are many ways to organize the elements: arrays are just one of them

- Each way is implementing some *data structure**.

There is no “best container”:

- Each is best suited to different problems

The 3 main properties of containers:

- **Homogeneous X heterogeneous**: whether all elements are the same type.
- **Static X dynamic**: whether the number of elements is fixed.
- **Sequentiality**:
  - Sequential containers: elements stored by order, and are accessed by indices.
  - Non-sequential containers: elements stored by name or through relationships.

**A data structure* is a way of organizing data; a *structure* is just one of them.

Containers

The most common types (names vary among languages; some have several implementations for the same type)*:

- **Array / vector / matrix (1D or MD)**: C, C++, Fortran, IDL, Java, Python+Numpy, R
- **List**: C++, Python, IDL (≥8), Java, R, Perl**
- **Map / hash / hashtable / associative array / dictionary**: C++, Python, IDL (≥8), Java, R***, Perl
- **Set**: C++, Python, Java, R
- **Tree / heap**: C++, Python, Java
- **Stack**: C++, Python, Java
- **Queue**: C++, Python, Java

*Listed only when the structure is part of a language's standard library.

**A Perl array is more like a list than an array.

***Which in R are also called *named lists.*

Arrays - definition

The simplest container.

A sequential set o elements, organized **regularly**, in 1D or more (MD).

Not natively present in some recent languages (Perl, Python without Numpy).

Sometimes called **array** only when more than 1D, being called **vector** in the 1D case.

Arrays - characteristics

Homogeneous (all elements must be the same **type**)

Static (cannot change the number of elements)

- "Dynamic arrays" are actually creating new arrays, and throwing away the old ones on resize (which is inefficient).

Sequential (elements stored by an order)

Organized in 1D or more (MD).

Element access through their indices (sequential integer numbers).

Usually, **the most efficient container for random and sequential access.**

Provide the means to do vectorization (do operations on the whole array, or parts of the array, with a single statement).

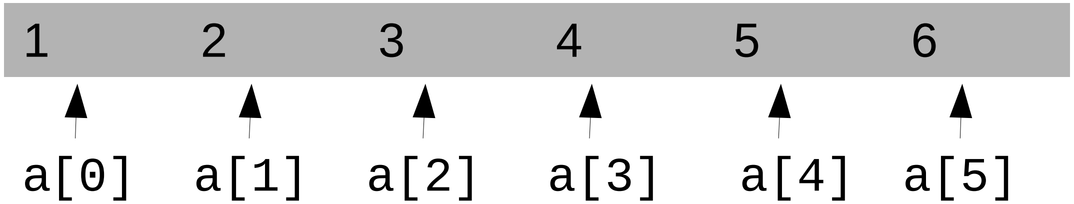
- 1D arrays are common.
- MD arrays are often awkward (2D may not be so bad): **IDL and Python+Numpy have high level MD operations**.

Internally all elements are **stored as a 1D array, even when there are more dimensions** (memory and files are 1D).

- **They are always regular** (each dimension has a constant number of elements).

Arrays

1D



Ex.:

```
IDL> a=bindgen(6)+1
```

Generates an array of type ***byte***, with 6 elements, valued 1 to 6.

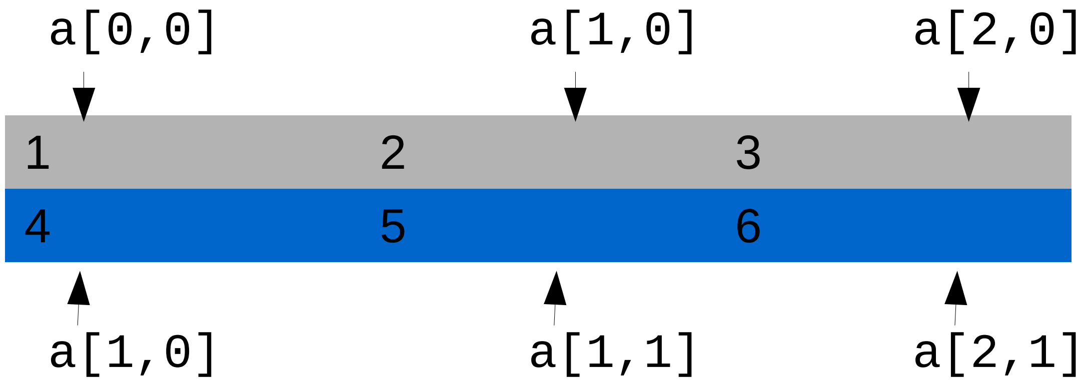
```
IDL> help,a
A               INT       = Array[6]
```

```
IDL> print,a
       1       2       3       4       5       6
```

In IDL, indexes start at 0 (other languages may start at 1 or at arbitrary numbers)

Arrays

2D



Ex.:

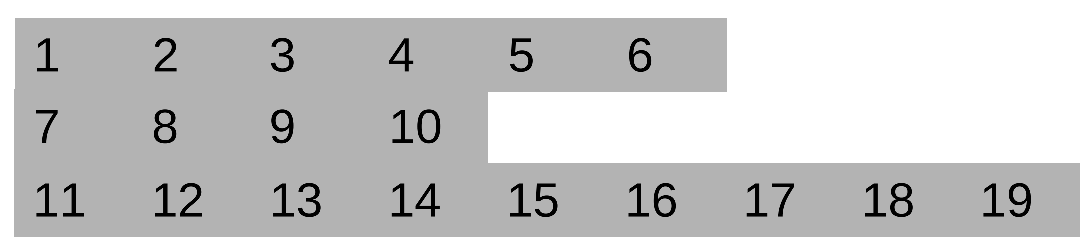
```
IDL> a=bindgen(3,2)+1
```

Generates an array of type ***byte***, with 6 elements, in 3 columns by 2 rows, valued 1 to 6.

```
IDL> help,a
A               INT       = Array[3, 2]

IDL> print,a
       1       2       3
       4       5       6
```

Must be regular: cannot be like



Arrays

3D is usually thought, graphically, as pile of “pages”, each page being a 2D table. Or as a brick. Ex:

```
IDL> a=bindgen(4,3,3)
```

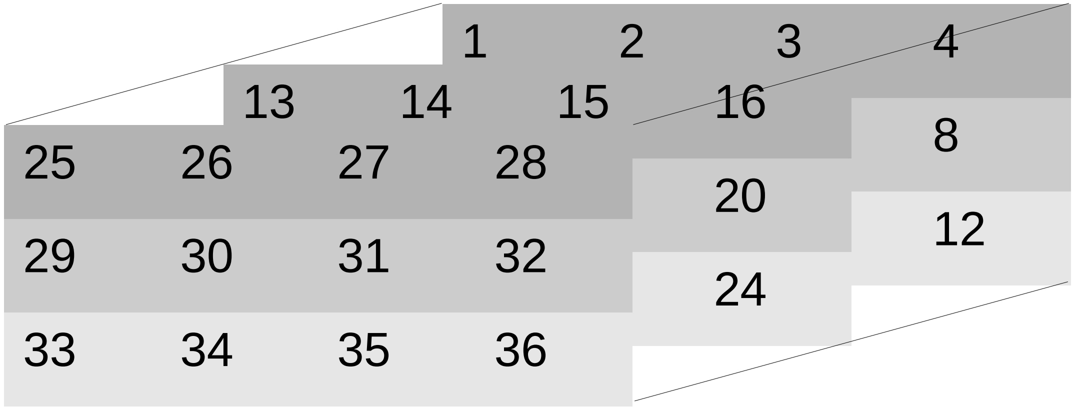
Generates an array of type ***byte***, with 36 elements, over 4 columns, 3 rows, 3 “pages”, valued 0 to 35.

```
IDL> help,a
A               BYTE      = Array[4, 3, 3]

IDL> print,a
   0   1   2   3
   4   5   6   7
   8   9  10  11

  12  13  14  15
  16  17  18  19
  20  21  22  23

  24  25  26  27
  28  29  30  31
  32  33  34  35
```

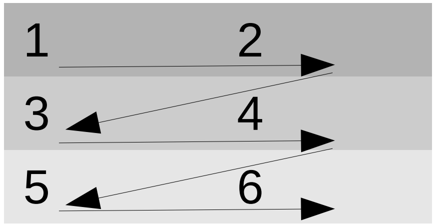


Beyond 3D, graphical representations get awkward (sets of 3D arrays for 4D, sets of 4D for 5D, etc.)

Arrays – MD storage

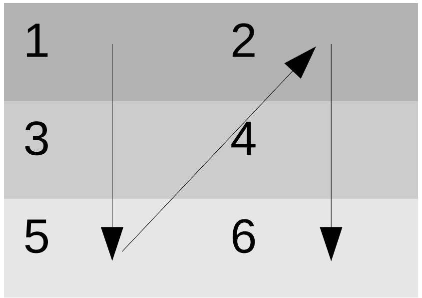
Internally, they are always 1D

The dimensions are scanned sequentially. Ex (2D): a[2,3] - 6 elements:



1)
`a[0,0] a[1,0] a[2,0] a[0,1] a[1,1] a[2,1]`
Memory position:
`0 1 2 3 4 5`

or



2)
`a[0,0] a[0,1] a[1,0] a[1,1] a[2,0] a[2,1]`
Memory position:
`0 1 2 3 4 5`

Each language has its choice of dimension order:

Column major – first dimension is contiguous (1 above): **IDL**, Fortran, R, Python+Numpy
Row major – last dimension is contiguous (2 above): C, C++, Java, Python+Numpy

Note that languages / people may differ in the use of the terms ***row*** and ***column***.

Graphically, usually the “horizontal” dimension (shown over a line) can be either the first of the last. Usually the horizontal dimension is the contiguous.

Arrays – basic usage

Access to individual elements, through the M indices (MD), or single index (MD or 1D). Ex:

```
IDL> a=dindgen(4)
IDL> b=dindgen(2,3)
IDL> help,a
A               DOUBLE    = Array[4]
IDL> help,b
B               DOUBLE    = Array[2, 3]
IDL> print,a
       0.0000000       1.0000000       2.0000000       3.0000000
IDL> print,b
       0.0000000       1.0000000
       2.0000000       3.0000000
       4.0000000       5.0000000
IDL> print,a[2]
       2.0000000
IDL> print,a[-1]
       3.0000000
IDL> print,a[-2]
       2.0000000
IDL> print,a[n_elements(a)-2]
       2.0000000
IDL> print,b[1,2]
       5.0000000
IDL> print,array_indices(b,5)
           1           2
IDL> print,b[5]
       5.0000000
```

Return arrays of **doubles** where each element has the value of its index. (→ `a=dindgen(4)`)

Negative indices are counted from the end (IDL≥8): -1 is the last element, -2 the one before the last, etc. (→ `print,a[-1]`)

Elements in MD arrays can also be accessed through their 1D index. (→ `print,b[5]`)

Arrays – basic usage

Accessing slices: regular subsets, 1D or MD, contiguous or not. Ex:

```
IDL> b=bindgen(4,5)
IDL> print,b
   0   1   2   3
   4   5   6   7
   8   9  10  11
  12  13  14  15
  16  17  18  19
IDL> c=b[1:2,2:4]
IDL> help,c
C               BYTE      = Array[2, 3]
IDL> print,c
   9  10
  13  14
  17  18
IDL> print,b[*,0:2]
   0   1   2   3
   4   5   6   7
   8   9  10  11
IDL> print,b[1:2,0:-1:2]
   1   2
   9  10
  17  18
IDL> print,b[1,2:0:-1]
   9
   5
   1
```

Elements from columns **1 to 2**, from lines 2 to 4 (for `c=b[1:2,2:4]`)

All columns, lines 0 to 2 (for `print,b[*,0:2]`)

Columns **1 to 2**, lines **0 to last (-1)**, **every second line** (stride 2) (for `print,b[1:2,0:-1:2]`)

Stride can be negative, to take elements in reverse order. (for `print,b[1,2:0:-1]`)

Arrays – should I care whether they are row/column major?

For most light, simple use, it does not matter.

When does it matter?

1) **Vector operations**: to select contiguous elements, to use single index for MD arrays.

2) **Mixed language / data sources:**

- When calling a function from another language, accessing files / network connections between different languages.

Arrays – should I care whether they are row/column major?

3) Efficiency:

If an array has to be scanned, it is more efficient (**specially in disk**) to do it in the same order used internally.
Ex: to run through all the elements of this column major array:

a[0,0] (a[0]) : 1	a[1,0] (a[1]) : 2
a[0,1] (a[2]) : 3	a[1,1] (a[3]) : 4
a[0,2] (a[4]) : 5	a[1,2] (a[5]) : 6

In the same order used internally:

```
for j=0,2 do begin
  for i=0,1 do begin
    k=i+j*2
    print,i,j,k,a[i,j]
    do_some_stuff,a[i,j]
  endfor
endfor
```

i	j	**k**	a[i,j]
0	0	**0**	1
1	0	**1**	2
0	1	**2**	3
1	1	**3**	4
0	2	**4**	5
1	2	**5**	6

No going back and forth (shown by variable **k**).

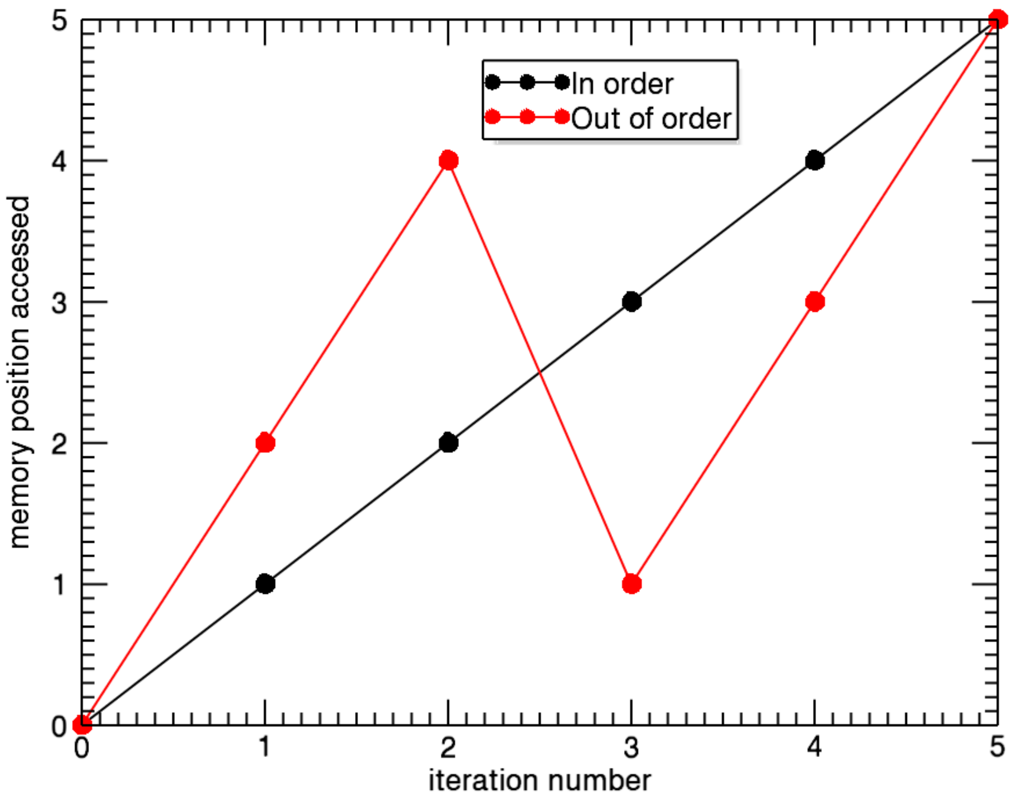
Arrays – should I care whether they are row/column major?

Reading out of order:

```
for i=0,1 do begin
  for j=0,2 do begin
    k=i+j*2
    print,i,j,k,a[i,j]
    do_some_stuff,a[i,j]
  endfor
endfor
```

i	j	**k**	a[i,j]
0	0	**0**	1
0	1	**2**	3
0	2	**4**	5
1	0	**1**	2
1	1	**3**	4
1	2	**5**	6

Lots of going back and forth:

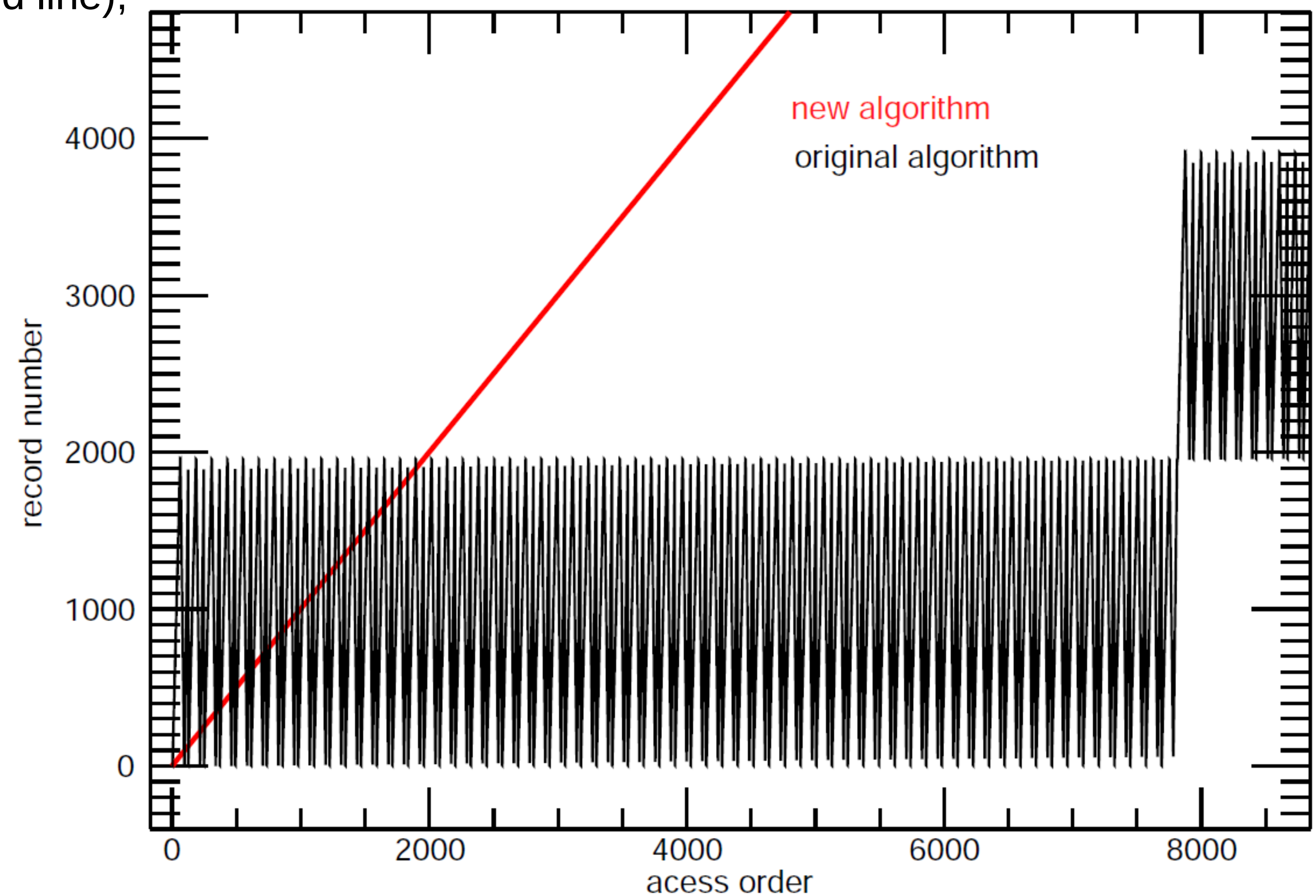


Arrays – should I care whether they are row/column major?

One real life example

The original code read through disk out of order, taking ~1h to run (black line).

When reading in order (red line), the code ran in ~3 min.



How to make an “irregular array”

If an array stores pointers, each element can point to anything, regardless of what the other elements point to:

```
IDL> arr[1]=ptr_new('banana')
IDL> arr=ptrarr(3)
IDL> arr[0]=ptr_new([1,2,3])
IDL> arr[1]=ptr_new([90,21])
IDL> arr[2]=ptr_new([18,49,37,84,93])
IDL> for i=0,2 do print,i,':',*arr[i]
       0:        1       2       3
       1:       90      21
       2:       18      49      37      84
93
IDL> arr[1]=ptr_new('banana')
IDL> for i=0,2 do print,i,':',*arr[i]
       0:        1       2       3
       1:banana
       2:       18      49      37      84
93
```

Lists - definition

Elements stored **sequentially**, accessed by their indices

- **Similar to 1D arrays.**

Unlike arrays, lists are dynamic, and, in IDL, heterogeneous.

Ex:

```
IDL> l=list()
IDL> l.add,2
IDL> l.add,[5.9d0,7d0,12d0]
IDL> l.add,['one','two']
IDL> help,l
L               LIST  <ID=1  NELEMENTS=3>
IDL> print,l
       2
       5.9000000       7.0000000       12.000000
one two
IDL> l.remove,1
IDL> print,l
       2
one two
IDL> l.add,bindgen(3),1
IDL> print,l
       2
   0   1   2
one two
```

Creates an empty list

Elements added to the list

Removes element from position **1**.
If position unspecified, the last element is removed.

Add element to position **1**. When position is unspecified, added to the end of the list.

Lists - characteristics

Efficient to add / remove elements, from any place in the list.

- Usually elements are added / removed to the end by default.

Most appropriate when

- The number of elements to be stored is not known in advance.
- The types / dimensions of the elements are not known in advance.
- When there will be many adds / removals of elements.

Lists – application examples

Easy storage of “non-regular” arrays.

Applications where each element in the list contains a different number of elements:

- Elements of
  - Asteroid families
  - Star / galaxy clusters
  - Planetary / stellar systems

- Neighbors of objects (from clustering / classification algorithms)
  - Observations / model results
  - Different number of observations for each object
  - Different number of sources found on each observation
  - Different number of objects used in each model

- Non regular grids
  - Model parameters (models are calculated for different values of each parameter)
  - Grids with non-regular spacing
  - Models with different numbers of objects / species

Lists – application examples

Easy storage of “non-regular” arrays. Exs:

```
IDL> l=list()
IDL> l.add,[1.0d0,9.1d0,-5.2d0]
IDL> l.add,[2.5d0]
IDL> l.add,[-9.8d0,3d2,54d1,7.8d-3]
IDL> print,l
       1.0000000       9.1000000      -5.2000000
       2.5000000
      -9.8000000       300.00000       540.00000    0.0078000000
IDL> a=l[2]
IDL> print,a
      -9.8000000       300.00000       540.00000    0.0078000000
```

Hashes / Dictionaries - characteristics

Similar to structures: store **values** by names (**keys**).

Unlike structures, **keys can be any data type** (most often used: strings, integers, reals).

Unlike indices (arrays and lists), **keys are not sequential.**

Unlike structures, dictionaries are dynamic: elements can be freely and efficiently added / removed.

- Dictionaries are to structures as lists are to 1D arrays.

May be heterogeneous – both keys and values can have different types / dimensions.

Elements may not be stored in order:

- The order the keys are listed may not be the same order in which they were put into the dictionary.

Find out whether a key is present, and retrieve the value from a key are operations that take **constant time**: It does not matter (usually) whether the dictionary has 10 or 1 million elements.

Hashes / Dictionaries - characteristics

Similar to structures: store **values** by names (**keys**).

Unlike structures, **keys can be any data type** (most often used: strings, integers, reals).

Unlike indices (arrays and lists), **keys are not sequential.**

Unlike structures, dictionaries are dynamic: elements can be freely and efficiently added / removed.

- Dictionaries are to structures as lists are to 1D arrays.

May be heterogeneous – both keys and values can have different types / dimensions.

Elements may not be stored in order:

- The order the keys are listed may not be the same order in which they were put into the dictionary.

Find out whether a key is present, and retrieve the value from a key are operations that take **constant time**: It does not matter (usually) whether the dictionary has 10 or 1 million elements.

- Key/value lookup does not involve searches.
- Like a paper dictionary, a paper phone book, or the index in a paper book.

In IDL 8.0 to 8.2.3, there is only one type: **hash**.

IDL 8.3 also has **orderedhash** and **dictionary**.

Hashes – basic use:

```
IDL> h=hash()
IDL> h['one']=[9.0,5.8]
IDL> h[18.7]=-45
IDL> h[10]=bindgen(3,2)
IDL> help,h
H               HASH  <ID=1  NELEMENTS=3>
IDL> print,h
10:    0   1   2 ...
one:       9.00000      5.80000
18.7000:      -45
IDL> print,h[10]
   0   1   2
   3   4   5
IDL> print,h.keys()
      10
one
      18.7000
IDL> print,h.values()
   0   1   2   3   4   5
      9.00000      5.80000
     -45
IDL> print,h.haskey('two')
           0
IDL> h.remove,'one'
IDL> print,h.haskey('one')
           0
```

Creates an empty dictionary (hash)

Add values to it

Hashes - examples

Storing elements by a useful name, to avoid keep searching for the element of interest. Ex: Storing several spectra, by the target name:

```
spectra=hash()
foreach el, files do begin
  read_spectrum,el,spectrum_data
  spectra[spectrum_data.target]=spectrum_data
endforeach
```

Which would be convenient to use:

```
IDL> help,h
H               HASH  <ID=1  NELEMENTS=3>

IDL> print,h
HR21948: { HR21948       5428.1000       5428.1390   5428.1780  5428.2170 ...
HR5438: { HR5438         5428.0000       5428.0390   5428.0780  5428.1170 ...
HD205937: { HD205937     5428.1000       5428.1390   5428.1780  5428.2170 ...

IDL> help,h['HR5438']
** Structure <90013e58>, 7 tags, length=4213008, data length=4213008, refs=6:
   TARGET          STRING    'HR5438'
   WAVELENGTH      DOUBLE    Array[1024]
   FLUX            DOUBLE    Array[1024]
   DATE            STRING    '20100324'
   FILE            STRING    'spm_0049.fits'
   DATA            DOUBLE    Array[512, 1024]
   HEADER          STRING    Array[142]
```

Hashes - examples

A lot of freedom in key choice:

- Strings are arbitrary, without the character limitations in structure fields (which cannot have whitespace or special symbols): **`-+*/\()[]{} ,”'`**.

- Special characters commonly appear in useful keys:
  - File names (**`some-file.fits`**)
  - Object names (**`alpha centauri, 433 Eros, 2011 MD`**)
  - Catalog identifier (**`PNG 004.9+04.9`**)
  - Object classification (**`[WC6],R*`**), etc.

- Non-strings are often useful:
  - Doubles – Julian date, wavelength, coordinates, etc.
  - Non consecutive integers, not starting at 0: Julian day, catalog number, index number, etc.

New types

Starting in IDL 8.3:

orderedhash

- Just like a regular hash, but preserver the order of the elements.

dictionary

- **J**ust like a regular hash, but keys must be strings, following the same rules as IDL variables:
  - Case-insensitive
  - No spaces or special characters
  - Cannot start with a number

- So that values can be accessed like structure fields:

```
IDL> d=dictionary()
IDL> d.nobjects=3
IDL> d.temperatures=[18.5,20.98,200.46]
IDL> d
{
    "NOBJECTS": 3,
    "TEMPERATURES": [18.500000, 20.980000, 200.46001]
}
IDL> d.nobjects=4
IDL> d.temperatures=[18.500000, 20.980000, 200.46001,23.6]
```

Other containers

Structures are usually implemented as types, but are also containers – **heterogeneous, static and non sequential**:

```
** Structure <9019c628>, 6 tags, length=64, data length=58, refs=2:
   ELEMENT          STRING    'argon'
   INTENSITY        DOUBLE           98.735900
   WIDTH            DOUBLE        0.0087539000
   ENERGY           DOUBLE           12.983800
   IONIZATION       INT              3
   DATABASE         STRING    'NIST Catalog 12C'
   WAVELENGTH       DOUBLE           6398.9548
```

Hashes are to structures (both non sequential) as lists are to arrays (both sequential): **the former is the dynamic version of the latter.**

Arrays, lists, structures and dictionaries are the 4 basic containers.

- Most others are specializations of these 4.

Container choice – lists x arrays

Lists and arrays store elements ordered by index. They share many uses.

Differences:

- Lists are dynamic, 1D and may be heterogeneous.
- Arrays are static, homogeneous, and may be more than 1D.

Usually,

- Lists are chosen when one needs:
  - “non regular arrays”
  - add/remove elements (particularly when the number of elements to store is not known in advance).
  - elements that are not scalar, or not of the same type.
- Arrays are more convenient when one needs:
  - More than 1D
  - vector operations
  - make sure that elements are scalar and of the same type

Container choice – structures x dictionaries

Structures and dictionaries store elements by name. They share many uses.

Main difference:

- Dictionaries are dynamic

- Structures are static

Usually,

- Dictionaries are more convenient when:
  - The keys / types are not known in advance
  - The values may have to change type / dimensions
  - Adding removing fields will be necessary
  - Keys are not just simple strings

- Structures are more convenient:
  - To put them into arrays, to do vector operations
  - To enforce constant type / dimensions of values

Vectorization – Why?

The programmer only writes high level operations:

```
IDL> a=dindgen(4,3,2)
IDL> b=a+randomu(seed,[4,3,2])*10d0
IDL> help,a,b
A               DOUBLE    = Array[4, 3, 2]
B               DOUBLE    = Array[4, 3, 2]
IDL> c=a+b
IDL> d=sin(c)
IDL> help,c,d
C               DOUBLE    = Array[4, 3, 2]
D               DOUBLE    = Array[4, 3, 2]
IDL> A=dindgen(3,3)
IDL> y=dindgen(3)
IDL> x=A#y ;Matrix product of matrix A (3,3) and vector y (3)
IDL> help,y,A,x
Y               DOUBLE    = Array[3]
A               DOUBLE    = Array[3, 3]
X               DOUBLE    = Array[3]
```

IDL may even do vector operations in parallel.

1D x MD indexing

In an array with more than 1 dimension (MD), elements can be selected by one index per dimension:

```
IDL> a=bindgen(4,3)*2
IDL> print,a
       0       2       4       6
       8      10      12      14
      16      18      20      22
IDL> print,a[1,2]
      18
```

Or by just one index, which is the order in which that element is stored in the array:

```
IDL> print,a[9]
      18
IDL> print,array_indices(a,9)
           1           2
```

MD->1D conversion:

```
IDL> adims=size(a,/dimensions)
IDL> print,adims
           4           3
IDL> print,a[1+adims[0]*2]
      18
```

Arrays as indices

Selecting multiple elements with array expressions

- When an array is used as indices to another array, the result has the same dimension as the index array:

```
IDL> a=bindgen(4,3)*2
IDL> print,a[[0,1,3,5]]
       0       2       6      10
IDL> print,[[0,1],[3,5]]
       0       1
       3       5
IDL> print,a[[[0,1],[3,5]]]
       0       2
       6      10
```

→ 1D array, 4 elements: 0,1,3,5 from **a**

→ 2D index array (of 1D indices), 4 elements

→ 2D array, 4 elements from **a**, given by the (1D) indices above.

- When each dimension receives an index array, these arrays must have the same shape. The result has this shape, with the elements selected by the corresponding indices

:

```
IDL> print,a[[0,1],[3,5]]
      16      18
```

→ 1D array, 2 elements:
0: `a[0,3]`
1: `a[1,5]`

Mixed shape operations

Vector operations are not limited to arrays of the same shape:

- When a scalar is applied to an array, the result is an array of the same shape with the scalar applied to each element:

```
IDL> print,1+[0,3,4]
       1       4       5
```

- If two arrays of different shapes are used: the smaller lenght of each dimension is used; the remaining elements from the larger array are ignored:

```
IDL> b=[0,1,2]

IDL> c=[1,2,3,4]

IDL> print,b*c
       0       2       6

IDL> print,c*2
       2       4       6       8
```

Searching in arrays

Finding an array element by its properties is one of the most common operations. Easy with IDL's array functions:

•**Filters:**

```
w=where((spectrum.wavelength gt 4d3) and (spectrum.wavelength lt 6d3),/null)
spectrum=spectrum[w]
```

(selects only the elements in **`spectrum`** where the field **`wavelength`** is between 4d3 and 6d3)

```
spectrum=spectrum[where(finite(spectrum.flux),/null)]
```

(selects the elements in **`spectrum`** where **`flux`** Is not **NaN** or **infinity**)

•**Specific elements**

```
w=where(observations.objects eq 'HD3728',/null)
p=plot(observations[w].wavelength,observations[w].flux)
```

If this has to be done often, it may be better to put the elements into a hash, which is directly indexed by the name:

```
p=plot((observations['HD3278']).wavelength,(observations['HD3278']).flux)
```

Searching in arrays

- Elements nearest to some real number:
  - Usually necessary to find elements in arrays of reals, since there mey not be any elements with exactly the value being looked for:

Index where the minimum occurs

```
halpha=6562.8d0
!null=min(lines.wavelength-halpha,minloc,/absolute)
do_some_stuff,lines[minloc]
```

Searching for the minimum in absolute value

- Find a value in a monotonic sequence.
  - Ex: In a model, change the temperature in the grid cells located at a certain radius (**r_search**):

```
IDL> help,temperature,r,theta,phi,r_search
TEMPERATURE     DOUBLE    = Array[300, 100, 200]
R               DOUBLE    = Array[300]
THETA           DOUBLE    = Array[100]
PHI             DOUBLE    = Array[200]
R_SEARCH        DOUBLE    =        74.279000
IDL> print,minmax(r)
       17.485000       100.00000
IDL> w=value_locate(r,r_search)
IDL> print,w,r[w],r[w+1]
         205       74.058829       74.334799
IDL> temperature[w,*,*]=some_other_temperature
```

Returns the index where **r** (a sorted array) surrounds the value being searched for (**r_search**).

foreach loops (starting on IDL 8.0)

Operate on each element of an array, list or hash:

```
IDL> a=[1,4,9]
IDL> foreach element,a,ind do print,ind,element
                     0       1
                     1       4
                     2       9
IDL> h=hash()
IDL> h[1]=95
IDL> h['two']=[4,-9,1]
IDL> h[1.87]='something'
IDL> foreach value,h,key do print,key,':',value
      1.87000:something
       1:       95
two:        4      -9       1

foreach element,a,ind do begin
  print,ind,element
endforeach
```

Container methods (introduced in IDL 8.4)

Map: apply the same function to every element of the container

```
IDL> l=list()
IDL> l.add,[1,3,9]
IDL> l.add,[18,24]
IDL> l.add,!null
IDL> l.add,98
IDL> l
[
    [1, 3, 9],
    [18, 24],
    null,
    98
]
IDL> l.map(lambda(x:n_elements(x)))
[
    3,
    2,
    0,
    1
]
```

Lambda function: creates a one-line function right in the middle of the code.

http://www.exelisvis.com/docs/IDL_Variable.html
http://www.exelisvis.com/docs/LAMBDA.html

Container methods (introduced in IDL 8.4)

```
IDL> x=[10.0,-20.0,40.0,100.0]
IDL> x.mean()
       32.500000
IDL> x.max()
       100.00000
IDL> x.median()
       40.000000
```

http://www.exelisvis.com/docs/IDL_Variable.html

http://www.exelisvis.com/docs/IDL_Number.html

http://www.exelisvis.com/docs/IDL_String.html

Strings – definition

A **string** is a variable representing text, as a sequence (a string) of characters.

Every programming language has at least one standard variable type to represent and process strings.

It is one of the most often needed types, for everything. Exs:

- Inform the user
- File names
- Identifiers (elements, dates, names, programs, algorithms, objects, properties, etc.)
- File input and output (though not all data files are made with text)
- Building commands1
- Most databases and web applications are string-centric

Among the basic variable types strings are the most complex to process.

Processing strings is not only *prints* and *reads*.

Strings - encoding

What makes up a string?

- Computers only “know” numbers (in binary).
- Nothing makes the contents of a variable or file intrinsically text. They are only 0s and 1s.
- The mapping between binary numbers and text **is determined by the encoding**, just like integer and real numbers are also encoded into binary digits.
- **Most languages assume a specific encoding**; some have different types for different encodings, and some may use string objects that can produce different encodings.

In ancient times (1980s) encoding was always the same: **ASCII** (*American Standard Code for Information Interchange*):

- 1 byte (7 or 8 bits) per character - 2^8 (256) or 2^7 (128) different characters.
- A standard table defines which character is encoded by each number in the range 0-127:

String encodings - ASCII

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

String encodings - ASCII

Not all ASCII characters are visible (*printable*). Some are whitespace (space, tabs, etc.), other are some form o control character (null, CR, LF, etc.).

Zero is reserved for control, meaning either an empty string (made of only a zero), or, in some cases (C), the end of a string.

Characters 128-255 **are not in the ASCII standard**. The characters vary with the chosen ASCII extension.

ASCII is the simplest encoding in use:: characters always have the same size in memory (1 byte), and are easily read, processed and converted to/from numbers.

ASCII still is the most common encoding in scientific programming, but not the only one.

Line termination varies among systems. The most common choices:

- Unix-like systems (Linux, Max OS X): LF (**L**ine**F**eed; ASCII 10)
- Windows: CR (**C**arriage **R**eturn; ASCII 13) followed by LF (ASCII 10)
- Mac OS 9 and earlier: CR (ASCII 13)

ASCII does not mean the same as “text file”.

In recent years, **Unicode** encoding, in its many forms, is becoming more widespread.

String encodings - ASCII

Why not always use ASCII?

It is not enough. It does not contain, for instance:

- Modified characters (diacritical marks, cedilla)
- Math symbols (beyond the very basic **+ - . * / ^ ! % > < =**)
  - Ex: ℝ ℤ **∀ ∂ ∃ ∑ ∫ ∮ ± ≌ ≥ ≤ × ∞ ∇ ≠**
- Physical symbols
  - Ex: **Å µ** ☉ **⊕**
- Greek letters
- Other symbols
  - Ex: → ↔ **⇌ ⇛ € ª º £ ¥ ¿ ¡**
- Characters from other languages (including those of many symbols, such as the forms used for Chinese and Japanese).

String encodings - Unicode

How to overcome the ASCII limitations?

The only widely used standard today is **Unicode**.

Developed to be “the one code”, with “every” character from “every” language, with metadata (**data describing the characters).**

It is not immutable, additions are decided by the Unicode Consortium (http://www.unicode.org/).

String encodings - Unicode

The Unicode catalog has data about the characters, which are used in queries and to identify them, i**ncluding names and properties**: *printable*, numeric, alphanumeric, capital, blank, language, math, etc.

Exs:

Unicode Character 'LATIN CAPITAL LETTER A' (U+0041)

A

Name	LATIN CAPITAL LETTER A
Block	Basic Latin
Category	Letter, Uppercase [Lu]
Combine	0
BIDI	Left-to-Right [L]
Mirror	N
Index entries	Latin Uppercase Alphabet, Uppercase Alphabet, Latin Capital Letters, Latin
Lower case	**U+0061**
Version	Unicode 1.1.0 (June, 1993)

Unicode Character 'INTEGRAL' (U+222B)

∫

Name	INTEGRAL
Block	Mathematical Operators
Category	Symbol, Math [Sm]
Combine	0
BIDI	Other Neutrals [ON]
Mirror	Y
Index entries	Integral Signs, INTEGRAL
See Also	latin small letter esh **U+0283**
Version	Unicode 1.1.0 (June, 1993)

(results from http://www.fileformat.info/info/unicode/char/search.htm)

Strings – Unicode support

Languages vary widely

- **Do not know Unicode** (on;y use ASCII): C, Fortran
- **Use ASCII natively** (including for sourcecode), but have some variable types and libraries to to process Unicode: C, C++, **IDL**, R:
-
```
IDL> maçã=1

maçã=1
    ^
% Syntax error.
IDL> some_string='maçã'
IDL> print,some_string
maçã
IDL> iplot,/test,title='Temperature (°C)'
```

- **Use Unicode natively** (including in sourcecode), and have extensive Unicode string support: Java, Python, Perl

Often (even when Unicode can be used in sourcecode), Unicode characters are written through ASCII with escape codes:

`IDL> p=plot(/test,title='!Z(00C5,222B)')` produces **Å∫**

Strings – basic processing

Most common operations

- **Concatenation**

```
IDL> a= 'some'
IDL> b=a+' string'
IDL> help,b
B               STRING    = 'some string'
```

- **Sorting**

```
IDL> help,a,b
A               STRING    = 'some'
B               STRING    = 'some string'
IDL> print,b gt a
   1
IDL> c=[a,b,'9','Some',' some','some other string']
IDL> print,c[sort(c)],format='(A)'
 some
9
Some
some
some other string
some string
```

Strings – basic processing

- **Logical value:**

Empty string (*null string*) is false, the rest is true:

```
IDL> c=''

IDL> if c then print,'c is not empty string' else print,"c is null string ('')"
c is null string ('')

IDL> c='a'

IDL> if c then print,'c is not empty string' else print,"c is null string ('')"
c is not empty string
```

Whitespace is not the same as empty string:

```
IDL> c=' '

IDL> if c then print,'c is not empty string' else print,"c is null string ('')"
c is not empty string
```

Strings – basic processing

- **Substrings**

```
IDL> print,strmid('abcdefg',3,2)
de
```

In IDL 8.4:

```
IDL> a='abcdefg'
IDL> a.substring(2,5)
cdef
```

- **Search for characters or substrings**

```
IDL> print,strpos('abcdefg','de')
           3
IDL> a='abcdefg'
IDL> a.indexof('d')                (IDL 8.4)
           3
```

Strings – basic processing

- **Others**

```
IDL> print,strlen('1234567')
           7

IDL> print,strlen(' 1234567 ')
           9

IDL> help,strtrim(' 1234567 ',2)
<Expression>    STRING    = '1234567'

IDL> print,strupcase('abcdEF')
ABCDEF

IDL> print,strjoin(['a','b','c'],'~')
a~b~c

IDL> a='some random text'
IDL> a.replace('random','specific')
some specific text

IDL> print,strsplit('temperature=19.8/K','=/',/extract),format='(A)'
temperature
19.8
K
```

Measuring string length includes whitespace.

(IDL 8.4)

Strings – creation from other types

Every time you see a number, it was converted to a string. Exs (DL):

Strings

```
IDL> print,[-1,0,9]
      -1       0       9

IDL> print,1d0,1B,1.0
       1.0000000   1      1.00000

IDL> help,string(1d0,1B,1.0)
<Expression>    STRING    = '       1.0000000   1      1.00000'
```

```
IDL> printf,unit,dblarr(3,4,3)
```

Puts variables in a file, as strings

Strings – explicit formatting

Often, the default way a string is created from a variable is not adequate (number of digits, use of scientific notation, spacing, etc.)

In such cases, one must specify how to create the string (by a format).

Each language has its way to specify a format, but there are two common standards: C-like and Fortran-like. IDL understands both types.

Strings – explicit formatting

Fortran style

No explicit format (default)

```
IDL> print,1d0+1d-9
       1.0000000

IDL> print,1d0+1d-9,format='(E16.10)'
1.0000000010E+00

IDL> print,'x=',1d0+1d-9,format='(A0,F16.13)'
X= 1.0000000010000
```

C (“*printf*”) style

```
IDL> print,format='(%"x=%16.10e")',1d0+1d-9
x=1.0000000010e+00
```

Strings - Fortran-style formatting

(just the main specifiers)

```
IDL> print,'x=',1d0+1d-9,format='(A0,F16.13)'
X= 1.0000000010000
```

Code	Meaning	Example(s)
`A`	String	`'(A)', '(A10)'`
`I`	Integer (decimal)	`'(I)', '(I10)','(-I2)'`
`B`	Integer (binary)	`'(B)', '(B0)'`
`Z`	Integer (hexadecimal)	`'(Z)', '(Z10)'`
`O`	Integer (octal)	`'(O)', '(O10)'`
`F`	Real (fixed point)	`'(F)','(F5.2)'`
`E, D`	Real (floating point)	`'(E)','(D16.10)'`
`G`	Real (fixed or floating, depending on value)	`'(G)','(G10)'`
`“”`	String literal	`'(“x=”,I10)'`
`X`	blanks	`'(A,10X,I)'`

There are modifiers for signs, exponents, leading zeros, line feed, etc.

Strings – C-style formatting (*printf*)

(just the main specifiers)

String with fields to be replaced by values, marked by codes with **%**

```
IDL> print,format='(%"x=%16.10e")',1.98549d-8
x=1.9854900000e-08
```

Code	Meaning	Eample(s)
`d,i`	Integer, decimal (*int*)	`%d, %5d, %+05d`
`u`	Integer, unsigned (*unsigned int*)	`%u, %7u`
`f,F`	Real, fixed-point (*double, float*)	`%f, %13.6f`
`e,E`	Real, floating point (*double, float*)	`%e, %16.10e`
`g,G`	Real, either fixed or floating point, depending on value (*double, float*)	`%g, %7.3G`
`x,X`	Integer, unsigned, hexadecimal (*unsigned int*)	%x, %10X
`o`	Integer, unsigned, octal (*unsigned int*)	`%o, %5o`
`s`	String (*string*)	`%s, %10s`
`c`	Character (*char*)	%c
`p`	Pointer – C-style - (*void **)	%p
%	Literal %	%%

There are modifiers for signs, exponents, leading zeros, line feed, etc.

Strings – implicit conversion to other types

```
IDL> help,fix(['17',' 17 ','17.1',' -17 ','9 8'])
<Expression>    INT       = Array[5]

IDL> print,fix(['17',' 17 ','17.1',' -17 ','9 8'])
      17      17      17     -17       9

IDL> print,double(['17',' 17 ','17.1',' -17 ','9 8'])
       17.000000       17.000000       17.100000      -17.000000
  9.0000000

IDL> readf,unit,a,b,c,d

IDL> a=0d0
IDL> b=0.0
IDL> c=0
IDL> reads,'17.1d0 18.9d0 -9',a,b,c
IDL> help,a,b,c
A               DOUBLE    =       17.100000
B               FLOAT     =      18.9000
C               INT       =      -9
```

Converts the string into the types of the variables **a,b,c,d**

Strings – conversion to other types

When default conversion is not enough, a format can be specified

```
IDL> a=0d0
IDL> b=0.0
IDL> c=0
IDL> reads,'17.1d0 something 18.9d0,-9',a,b,c
% READS: Input conversion error. Unit: 0, File: <stdin>
% Error occurred at: $MAIN$
% Execution halted at: $MAIN$
```

Variables have to be created, to determine the types for the conversion

It did not work, because alone it does not know what to do with the **“something”.** Using a format:

```
IDL> reads,'17.1d0 something 18.9d0,-9',
a,b,c,format='(D6.1,11X,D6.1,1X,I)'

IDL> help,a,b,c
A               DOUBLE    =       17.100000
B               FLOAT     =      18.9000
C               INT       =      -9
```

The format instructed IDL to read a double (**D6.1**), skip 11 characters (**11X**), read a double (**D6.1**), skip one character (**1X**), and read an integer (**I**).

Strings – other examples

- **Simple tests**:

```
IDL> str=['a.fits','a.FITS','a.fitsa','ab.fits','abc.fits']

IDL> print,strmatch(str,'*.fits')
   1   0   0   1   1

IDL> print,strmatch(str,'*.fits',/fold_case)
   1   1   0   1   1

IDL> print,strmatch(str,'*.fits*',/fold_case)
   1   1   1   1   1

IDL> print,strmatch(str,'?.fits')
   1   0   0   0   0

IDL> print,strmatch(str,'??.fits')
   0   0   0   1   0
```

Strings methods (introduced in IDL 8.4)

```
IDL> a='some string'
IDL> a.capwords()
Some String
IDL> a.replace('string','bananas')
some bananas
IDL> a.contains('some')
   1
```

http://www.exelisvis.com/docs/IDL_String.html

Regular expressions - definition

Regular expressions, (regex, regexp) are the most powerful tool to specify properties of strings.

Regex are a language, implemented similarly on most programming languages.

What are they for?

The interpreter (***regular expression engine***) gets the string and the expression, and determines whether the string *match* that expression.

In some cases, the interpreter can also inform which parts of the string match which part of the regex, and extract these parts.

Regular expressions – use cases

- **Separate parts of strings**
  - Find lines with names, values and comments, and extract these pieces:

Scalar with a comment (as in a FITS file):

```
'SLITPA  =              351.979 / Slit position angle'
```

1D array spanning several lines

```
'BAND_BIN_CENTER = (0.350540,0.358950,0.366290,0.373220,0.379490,
   0.387900,1.04598)'
```

Scalars in different formats:

```
'Total Mechanical Luminosity:                         1.5310E+03'
'resources_used.walltime=00:56:03'
```

Pieces of names:

```
'60.63  1.7836E-20  2.456   T    FeIX((3Pe)3d(2PE)4p_1Po-3s2_3p6_1Se)'
```

Dates, separating year, month, day, hour, minute, second:

```
'DATE-OBS= '2006-12-18         ' / universal date of observation'
'DATE_TIME = 2010-07-19T16:10:32'
'START_TIME = "2006-182T22:51:02.850Z"'
```

Regular expressions – use cases

- **Separate pieces of strings**
  - Extract pieces of files names, because they mean something about the file contents:

```
'spec/dec18s0041.fits'
'scam/dec18i0054.fits'
'15_7_mts_hm/pixselh_mr15.sav'
'15_7_mts_hw/pixselh_mr15.sav'
'16_3_mts_hw/pixselb_mr16.sav'
'readmodel5l_-1_0.00010000_1.0000_r05_030_08196_0.100000_0.05000000_10.00.eps'
```

- **Determine whether a string represents a number** (integer or real, fixed or floating point).

- **Locate identifiers in file contents. Exs:**
  - Catalog identifiers in the middle of the text
  - Web addresses (http, ftp, etc.)
  - File names
  - Form values
  - Data elements in text

Regular expressions – simple example

Ex: Determine which strings represent a date in the format **`yyyy-mm-dd`**:

```
IDL> strs=['20100201','2010-02-01','2010-2-1','aaaa-mm-dd','T2010-02-01J']
IDL> print,stregex(strs,'[0-9]{4}-[0-9]{2}-[0-9]{2}',/boolean)
   0   1   0   0   1
```

This regex means:

- 4 repetitions (**`{4}`**) of digits (characters in the range **`[0-9]`**),
- Followed by (**`-`**),
- Followed by 2 repetitions (**`{2}`**) of digits (**`[0-9]`**),
- Followed by (**`-`**),
- Followed by 2 repetitions (**`{2}`**) of digits (**`[0-9]`**).

A slightly more complex regex could match the 3 date formats above. It could also reject the last expression (which has extra characters before and after the date).

Regular expressions - rules

A regex with “normal”* characters specifies a string with those characters, in that order.

- Ex: **'J'** is a regex that matches any string containing **J**. '**JA'** is a regex that only matches strings containing '**JA**'.
- Exs. (IDL):

```
IDL> strs=['J','JJJJJ','aJA','j','aJa']

IDL> print,stregex(strs,'J',/boolean)
   1   1   1   0   1

IDL> print,stregex(strs,'JA',/boolean)
   0   0   1   0   0
```

*some characters have special meaning in regular expressions (shown ahead).

Regular expressions – special characters

These symbols have special meanings. To represent literally that symbol, is must be escaped with a \:

<table>
<tr><th>Symbol</th><th>Meaning</th><th>example</th><th>Match</th></tr>
<tr><td>\</td><td>Escape: the following character must be interpreted literally, not by its special meaning.</td><td>'\?'</td><td>'?, 'a?a'</td></tr>
<tr><td>.</td><td>Any character</td><td>'a.b'</td><td>'ajb', 'aab', 'abb', 'jafbc'</td></tr>
<tr><td>+</td><td>One or more repetitions of the preceding element.</td><td>'a+b'</td><td>'ab', 'aab', 'bab', 'baabh'</td></tr>
<tr><td>()</td><td>Subexpression: groups characters so that several of them are affected by the modifiers (like parenthesis in math).</td><td>'(ab)+c'</td><td>'abc', 'ababc', 'dabababcg'</td></tr>
<tr><td>*</td><td>Zero or more repetitions of the preceding element.</td><td>'a*b'</td><td>'ab','b','aab', 'caaabg'</td></tr>
<tr><td>?</td><td>Zero or one occurrence of the preceding. element</td><td>'a?b'</td><td>'b', 'ab', 'cabd', 'cbd'</td></tr>
<tr><td>|</td><td>Alternation: either one of the two elements.</td><td>'a|bc'</td><td>'ac', 'bc', 'jacd', 'jbcd'</td></tr>
<tr><td>{n}</td><td>Exactly n repetitions of the preceding element.</td><td>'a{2}b'</td><td>'aab', 'daaabg'</td></tr>
<tr><td>{n1,n2}</td><td>From n1 to n2 repetitions of the preceding element.</td><td>'a{1,2}b'</td><td>'ab', 'aab', 'aaab', 'gaaabbd'</td></tr>
<tr><td>^</td><td>Anchor: beginning of string.</td><td>'^ab'</td><td>'ab', 'abb'</td></tr>
<tr><td>$</td><td>Anchor: end of string.</td><td>'ab$'</td><td>'ab','aab'</td></tr>
<tr><td>[]</td><td>Value set (shown ahead)</td><td></td><td></td></tr>
</table>

Regular expressions – value sets

[] means a set a value, which may be:

- **A set of things to match.**
  - Ex: **'[abc]'** means any of the characters **a**,**b**,**c**: Ex. Matches: **'a'**,**'b'**,**'c'**,**'ab'**,**'ha'**.
- **A set of things not to match**
  - **'[^abc]'** means anything other than **a**, **b** ou **c**: Ex matches: **'d'**, **'jgs'**, **'gg'**.
- **Value ranges**
  - **'[0-9]'** any digit
  - **'[0-9a-zA-Z]'** any digit or letter
- **Value classes**
  - Special names for some types of values (in IDL, these come delimited by **[::]):**
  - ex: **'[[:digit:]]'** means the same as **'[0-9]'**.

Regular expressions – value classes

<table>
<tr><th>Class</th><th>meaning</th></tr>
<tr><td>`alnum`</td><td>Alphanumeric characters: 0-9a-zA-Z</td></tr>
<tr><td>`alpha`</td><td>Alphabetic chracters: a-zA-Z</td></tr>
<tr><td>`cntrl`</td><td>ASCII control characters (not printable, codes 1 to 31 and 127).</td></tr>
<tr><td>`digit`</td><td>Digits (decimal): 0-9</td></tr>
<tr><td>`graph`</td><td>Printable characters: ASCII 33 to 126 (excl. space).</td></tr>
<tr><td>`lower`</td><td>Lower case letters: a-z</td></tr>
<tr><td>`print`</td><td>Printable characters “imprimíveis” (visible plus space): ASCII 32 to 126.</td></tr>
<tr><td>`punct`</td><td>Punctuation: !"#$%&’()*+,-./:;<=>?@[\]^_‘{|}~</td></tr>
<tr><td>`space`</td><td>Whitespace: space, tab, vertical tab, CR, LF (ASCII 32 and 9-13).</td></tr>
<tr><td>`upper`</td><td>Capital letters: A-Z</td></tr>
<tr><td>`xdigit`</td><td>Hexadecimal digits: 0-9A-Fa-f</td></tr>
<tr><td>`<`</td><td>Beginning of the word (“word”meaning a sequence of non-space characters).</td></tr>
<tr><td>`>`</td><td>End of word.</td></tr>
</table>

These are just the main classes.

Regular expressions - examples

Determine whether a string represent a number.:

```
IDL> str=['9','-18',' 8.75','-8.1','.2','-.459','1.3E9','-9.8d7','a18.8d0','3.2f5']
```

•**Integers**:

```
IDL> intexpr='^[-+]?[0-9]+$'
```

Optional sign — `[-+]?`

1 or more digits — `[0-9]+`

```
IDL> print,stregex(str,intexpr,/boolean)
   1   1   0   0   0   0   0   0   0   0
```

•**Floating point:**

Fixed-point number or floating-point number (mantissa and exponent)

```
IDL> fpexpr='^[-+]?(([0-9]*\.?[0-9]+)|([0-9]+\.?[0-9]*))([eEdD][-+]?[0-9]+)?$'
```

Optional sign — `[-+]?`

0 or more digits, optionally followed by a period, plus 1 or more digits — `([0-9]*\.?[0-9]+)`

or

1 or more digits, optionally followed by a period, plus 0 or more digits — `([0-9]+\.?[0-9]*)`

Optional exponent: letter (e/d), followed by optional sign, foloowed by 1 or more digits — `([eEdD][-+]?[0-9]+)?`

```
IDL> print,stregex(str,fpexpr,/boolean)
   1   1   0   1   1   1   1   1   0   0
```

Regular expressions - extraction

Regular expressions can also be used **to extract pieces of the string, that matched pieces of the expression**.

Ex: Determine whether a string contains a date, in any of these formats

```
IDL> dates=['2011-01-31','2011 1 31','2011/01/31','something done on y2011m1d31 with something']
```

And extract the dates from the strings

```
IDL> expr='[0-9]{4}.[0-9]{1,2}.[0-9]{1,2}'
```

(4 digits)(any separator)(1 to 2 digits)(any separator)(1 to 2 digits)

```
IDL> print,stregex(dates,expr,/extract),format='(A)'
2011-01-31
2011 1 31
2011/01/31
2011m1d31
```

Now, how do we extract each piece (year, month, day)? One operation for each part?

- Could be, much a regex does it all.

Regular expressions - extraction

- In this case, to make for a smaller regex, we assume a simple format: (**`yyyy-mm-ddThh:mm:ss.fff`**).

```
IDL> str='Stuff observed on 2011-01-31T12:39:24.983 with some instrument'
IDL> expr='([0-9]{4})-([0-9]{2})-([0-9]{2})T([0-9]{2}):([0-9]{2}):([0-9]{2}\.[0-9]{3})'
```

(4 digits) **-** (2 digits) **-** (2 digits) **T** (2 digits) **:** (2 digits)**:** (2 digits**.**3 digits)

```
IDL> pieces=stregex(str,expr,/extract,/subexpr)
IDL> print,pieces,format='(A)'
2011-01-31T12:39:24.983      Whole match
2011                         First subexpr
01                           Second subexpr
31                           Third subexpr
12                           Fourth subexpr
39                           Fifth subexpr
24.983                       Sixth subexpr
IDL>
d=julday(pieces[2],pieces[3],pieces[1],pieces[4],pieces[5],pieces[6])
IDL> print,d,format='(F16.6)'
  2455593.027372
```

Some references

References:

The IDL Way, by David Fanning
http://www.idlcoyote.com/idl_way/idl_way.php

- Including *"My IDL Program Speed Improved by a Factor of 8100!!!"*
  http://www.idlcoyote.com/code_tips/slowloops.html

Characters vs. bytes
http://www.tbray.org/ongoing/When/200x/2003/04/26/UTF

The absolute minimum every software developer absolutely, positively must know about Unicode and character sets (no excuses!)
http://www.joelonsoftware.com/articles/Unicode.html

Unicode character search
http://www.fileformat.info/info/unicode/char/search.htm

Software Carpentry Videos on Regular Expressions:
http://software-carpentry.org/4_0/regexp/

This presentation is at
http://www.ppenteado.net/idl/intro

Some references

http://xkcd.com/208/