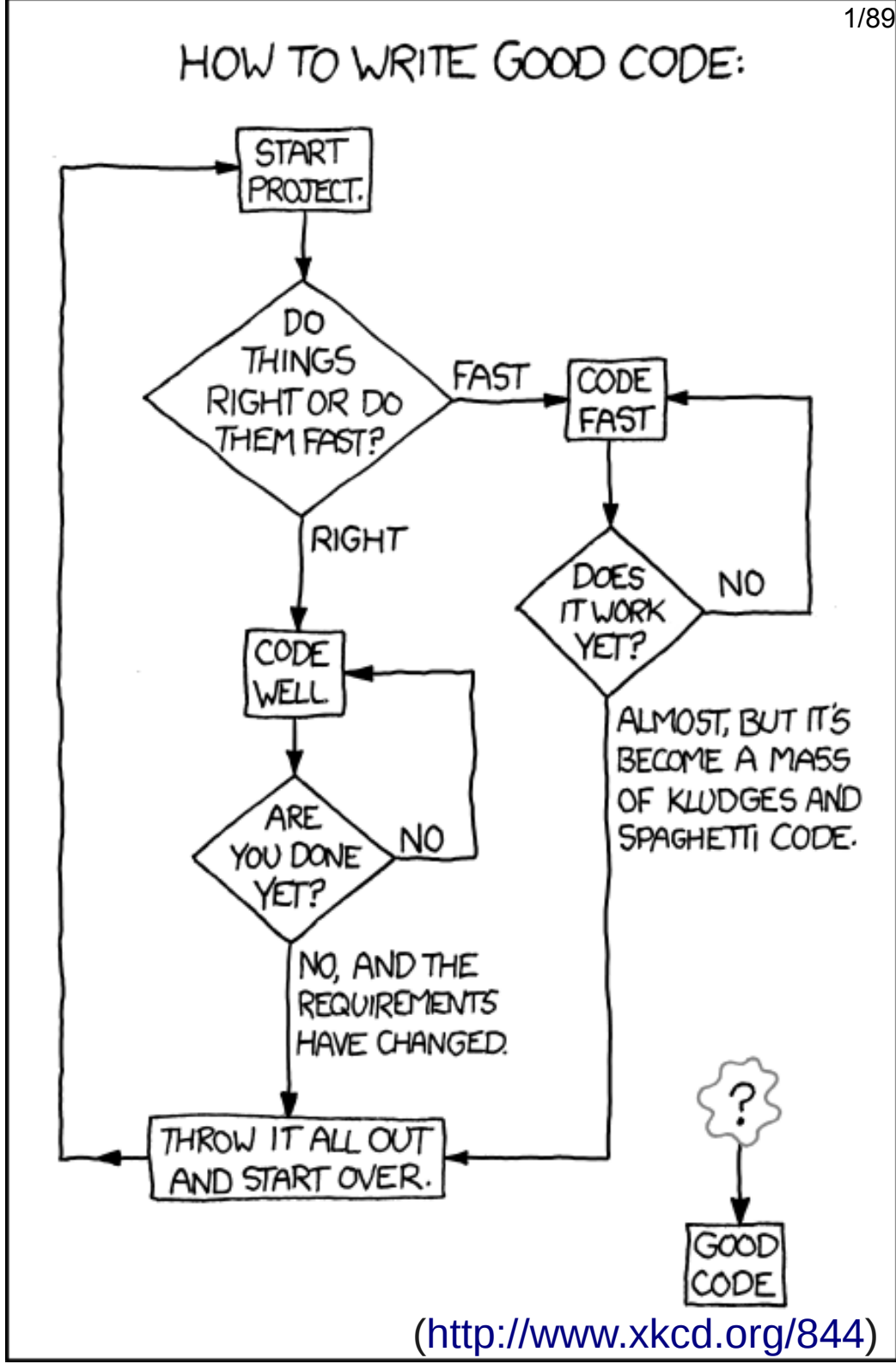
Programação em astronomia: indo além de loops e prints

4 - Contêiners

Paulo Penteado

http://www.ppenteado.net/pea



(http://www.xkcd.org/844)

Programa

1 – Slides em http://www.ppenteado.net/pea/pea01_linguagens.pdf
- Motivação
- Tópicos abordados
- Tópicos omitidos
- Opções e escolha de linguagens
- Uso de bibliotecas
- Referências

2 – Slides em http://www.ppenteado.net/pea/pea01_organizacao.pdf
- Organização de código
- Documentação
- IDEs
- Debug
- Unit testing

3 – Slides em http://www.ppenteado.net/pea/pea02_variaveis.pdf
- Tipos de variáveis
- Representações de números e suas conseqüências
- Ponteiros
- Estruturas
- Objetos

Programa

4 – Slides em http://www.ppenteado.net/pea/pea03_conteiners.pdf
- Contêiners
- Arrays
- Listas
- Mapas
- Outros contêiners
- Vetorização
- Escolha de contêiners

5 – Slides em http://www.ppenteado.net/pea/pea04_strings_io.pdf
- Strings
- Expressões regulares
- Arquivos

Contêiners

Apenas variáveis escalares não é suficiente.

Sempre é necessário usar **contêiners** - variáveis que armazenam várias instâncias de um ou mais tipos (os elementos).

Há muitas formas de organizar os elementos: arrays são apenas uma delas.

Cada forma de organizar os elementos é uma realização de alguma *estrutura de dados**.

Assim como em escolha de linguagens, **não existe "o melhor contêiner"**:

- Cada um foi desenhado para alguns tipos de problemas.

As três principais propriedades de contêneirs:

- **Homogêneos X heterogênos**: elementos têm ou não que ser do mesmo tipo.
- **Estáticos X dinâmicos**: o número de elementos contidos é ou não fixo.
- **Sequencialidade**:
  - Nos sequenciais, os elementos são armazenados ordenadamente, e são acessados por índice(s) que identificam os elementos pela sua ordem.
  - Nos não sequenciais, o acesso é feito por algum *nome* (chave, nome do campo).

* *Estrutura de dados* é uma forma de organizar dados: *estrutura* é apenas uma das formas.

Contêiners

Tipos mais comuns (nomes variam entre linguagens; algumas têm várias implementações do mesmo tipo)*:

- ***Array* / vetor / matriz (1D ou MD)**: C, C++, Fortran, IDL, Java, Python+Numpy, R
- **Lista**: C++, Python, IDL (≥8), Java, R, Perl**
- **Mapa / *hash* / *hashtable* / array associativo / dicionário**: C++, Python, IDL (≥8), Java, R***, Perl
- **Conjunto**: C++, Python, Java, R
- **Árvore / *heap***: C++, Python, Java
- **Pilha (*stack*)**: C++, Python, Java
- **Fila (*queue*)**: C++, Python, Java
- ***Linked list***: de mais baixo nível, às vezes usadas para implementar filas, pilhas e lists.

*Listadas apenas as linguagens que têm a estrutura em suas bibliotecas padrão

**O que Perl chama de array tem mais características de listas do que arrays.

***Que em R, estranhamente, são também chamadas de listas (*named lists*).

Arrays - definição

O contêiner mais simples, implementado até em velhas linguagens estáticas.

Um conjunto sequencial de elementos, organizados **regularmente**, em 1D ou mais.

Já não presente nativamente em algumas novas dinâmicas (Perl, Python sem Numpy).

Às vezes chamado de ***array*** só quando tem mais de 1D, e para 1D é chamado de **vetor**.

Arrays 2D às vezes são chamados de **matrizes**

- Em algumas linguagens (ex: R, Python+Numpy), ***matrix*** é diferente de arrays genéricos.

Arrays - características

Homogêneos (todos os elementos são do mesmo **tipo**)

Estáticos (não é possível mudar o número de elementos)
- Arrays dinâmicos (que permitem mudar o tamanho) estão gerando novos arrays e copiando os elementos, e descartando os antigos, o que é ineficiente.

Sequenciais (elementos armazenados em uma ordem)

Organizados em 1D ou mais (MD).

Acesso aos elementos através de seu(s) índice(s).

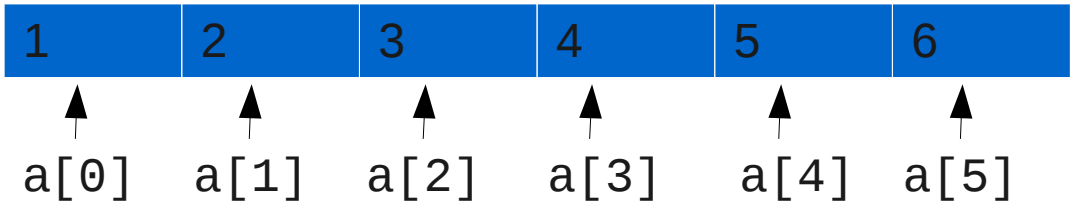
Em geral, **o contêiner mais eficiente para acesso aleatório e sequencial, e o que consome menos memória.**

São o principal meio de fazer vetorização (adiante).
- 1D é muito comum.
- MD é com freqüência desajeitado (2D ocasionalmente não é tão ruim): **apenas IDL e Python+Numpy têm arrays MD de alto nível** (adiante).

Internamente, todos os elementos são **armazenados em uma seqüência 1D, mesmo quando há mais de uma dimensão** (memória e arquivos são unidimensionais).
- **Em mais de 1D, são sempre regulares** (cada dimensão tem um número constante de elementos).

Arrays

1D



Ex. (IDL):

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

→ Gera um array de tipo ***byte***, com 6 elementos, com valores 1 a 6.

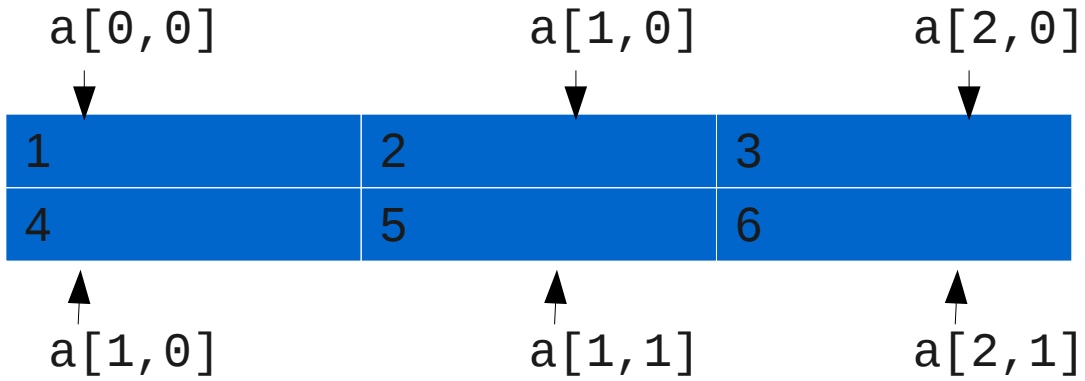
```
IDL> help,a
A               INT       = Array[6]
```

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

O mais comum é índices começarem em 0. Em algumas linguagens, pode ser escolhido.

Arrays

2D



Ex. (IDL):

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

Gera um array de tipo ***byte***, com 6 elementos, em 3 colunas por 2 linhas, com valores 1 a 6.

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

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

São regulares: não podem ser como

1	2	3	4	5	6			
7	8	9	10					
11	12	13	14	15	16	17	18	19

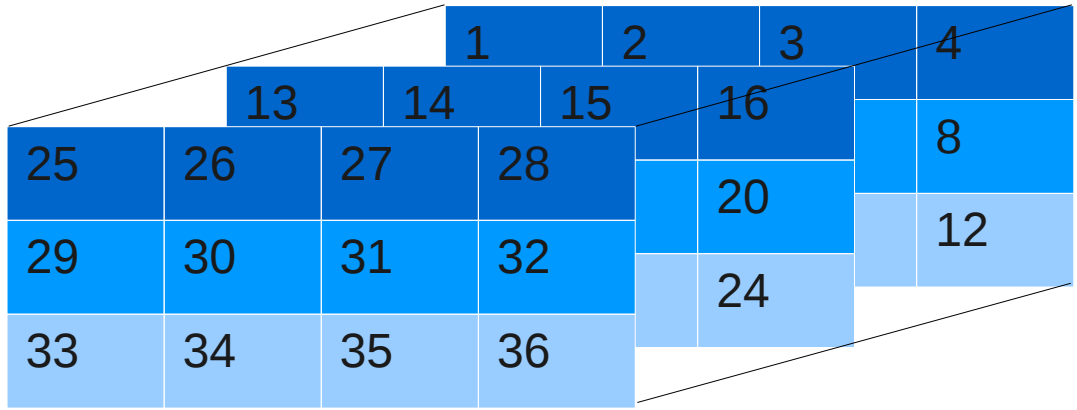
Arrays

3D costuma ser pensado, graficamente, como uma pilha de “páginas”, cada página sendo uma tabela 2D (ou um paralelepípedo). Ex. (IDL):

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

Gera um array de tipo ***byte***, com 36 elementos, em 4 colunas, 3 linhas, 3 “páginas”, com valores 0 a 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
```



Para mais que 3D, a imagem gráfica costuma ser conjuntos de pilhas 3D (para 4D), conjuntos de 4D (para 5D), etc.

Arrays - armazenamento MD

Se internamente são sempre seqüências 1D, como são armazenados arrays MD?

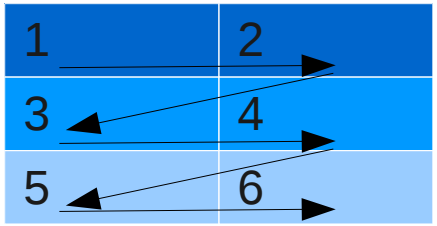
As várias dimensões são varridas ordenadamente. Ex (2D): a[2,3] - 6 elementos:

1)

```
a[0,0] a[1,0] a[2,0] a[0,1] a[1,1] a[2,1]
```

Posição na memória:

```
0      1      2      3      4      5
```



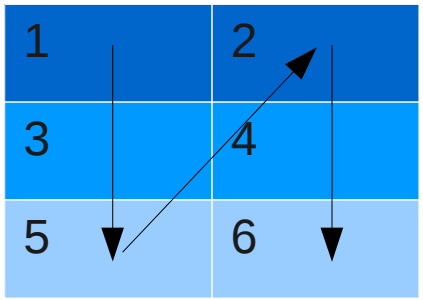
ou

2)

```
a[0,0] a[0,1] a[1,0] a[1,1] a[2,0] a[2,1]
```

Posição na memória:

```
0      1      2      3      4      5
```



Cada linguagem faz sua escolha de como ordenar as dimensões.

Column major - primeira dimensão é contígua (1 acima): IDL, Fortran, R, **Python+Numpy**
Row major - última dimensão é contígua (2 acima): C, C++, Java, **Python+Numpy**

Linguagens podem diferir no uso dos termos ***row*** and ***column***.

Graficamente, em geral a dimensão “horizontal” (mostrada ao longo de uma linha) pode ser a primeira ou a última. Normalmente, a dimensão horizontal é a contígua.

Arrays – uso básico

Acesso a elementos individuais, pelos M índices (MD), ou um único (MD ou 1D). Ex. (IDL):

```
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
```

Retornam arrays de **doubles** onde cada elemento tem o valor de seu índice; há versões para os outros tipos numéricos.

Índices negativos são contados a partir do final (Python+Numpy, R, IDL≥8): -1 é o último, -2 é o penúltimo, etc.

Elementos de arrays MD podem ser acessados também por seu índice 1D (IDL, Python+Numpy)

Arrays – uso básico

Acesso a fatias (*slices)*: conjuntos regulares*, 1D ou MD, contíguos ou não. Ex. (IDL):

```
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
```

Elementos das colunas **1 a 2**, das linhas 2 a 4

Todas as colunas, linhas 0 a 2

Colunas **1 a 2**, linhas **0 a última (-1)**, de **2 em 2** linhas (com um passo (*stride*) 2)

O passo pode ser negativo, para andar na ordem reversa

*Apenas com Numpy há fatias não regulares, através da escolha de passos.

Arrays – uso básico

Dimensões de tamanho 1 também pode contar: um array [1,n] não é o mesmo que um array [n]. Ex. (IDL):

```
IDL> e=bindgen(4)
IDL> help,e
E               BYTE      = Array[4]
IDL> print,e
   0   1   2   3
IDL> f=bindgen(1,4)
IDL> help,f
F               BYTE      = Array[1, 4]
IDL> print,f
   0
   1
   2
   3
IDL> g=bindgen(4,1)
IDL> help,g
G               BYTE      = Array[4]
IDL> print,g
   0   1   2   3
```

1D, 4 elementos

2D, 4 elementos: 1 coluna, 4 linhas

2D, 4 elementos: 4 colunas, 1 linha: equivalente a 1D, pois 1D é uma linha

Arrays não são o mesmo que matrizes: matrizes são 2D, e em algumas linguagens (Python+Numpy, R) há subclasses *matrix*, mais limitadas, mas otimizadas para álgebra linear. Em Java, há matrizes no pacote Jama, que provê funcionalidade 2D e álgebra linear.

A principal importância de arrays está em operações vetoriais, (adiante, onde estão os exemplos para usos de arrays).

Arrays - que importa se o array é *row* ou *column major*?

1) **Operações vetoriais** (adiante): para selecionar vários elementos contíguos.

2) **Operação entre linguagens diferentes:**

- Chamando rotinas de outras linguagens, acessando arquivos e conexões de rede escritos / a serem lidos em outras linguagens.

Arrays - que importa se o array é *row* ou *column major*?

3) Semântica de literais. Ex. (IDL):

```
IDL> a=[[1,2,3],[4,5,6]]

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

IDL> print,a
       1       2       3
       4       5       6

IDL> b=[[[1,2,3,4],[5,6,7,8],[9,10,11,12]],[[13,14,15,16],
[17,18,19,20],[21,22,23,24]]]

IDL> help,b
B               INT       = Array[4, 3, 2]

IDL> print,b
       1       2       3       4
       5       6       7       8
       9      10      11      12

      13      14      15      16
      17      18      19      20
      21      22      23      24
```

Arrays - que importa se o array é *row* ou *column major*?

4) Concatenações. Ex. (IDL):

```
IDL> a=[1,2,3]
IDL> b=[4,5,6]
IDL> c=[a,b]
IDL> help,c
C               INT       = Array[6]
IDL> print,c
       1       2       3       4       5       6
IDL> d=[[a],[b]]
IDL> help,d
D               INT       = Array[3, 2]
IDL> print,d
       1       2       3
       4       5       6
IDL> e=[[[a],[b]],[[-d]]]
IDL> help,e
E               INT       = Array[3, 2, 2]
IDL> print,e
       1       2       3
       4       5       6

      -1      -2      -3
      -4      -5      -6
```

Concatenamento na primeira dimensão (a da esquerda)

Concatenamento na segunda dimensão

Concatenamento na terceira dimensão

Concatenações com o próprio array (ex. **`a=[a,b]`**) criam um novo array, e copiam os valores para o novo.
O array (**a**) **não é redimensionado**: arrays são estáticos (não mudam de tamanho).

Arrays - que importa se o array é *row* ou *column major*?

5) Eficiência:

Se o array tem que ser percorrido, é mais eficiente (**especialmente em disco**) o fazer na mesma ordem usada internamente: o acesso é sequencial, sem idas e vindas.

Ex: para percorrer todos os elementos do array *column major*

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

Na mesma ordem de armazenamento, é uma passagem direta pela memória

```
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
```

Sem idas e voltas (mostrado pela variável **k**, que indica que posição na memória foi usada).

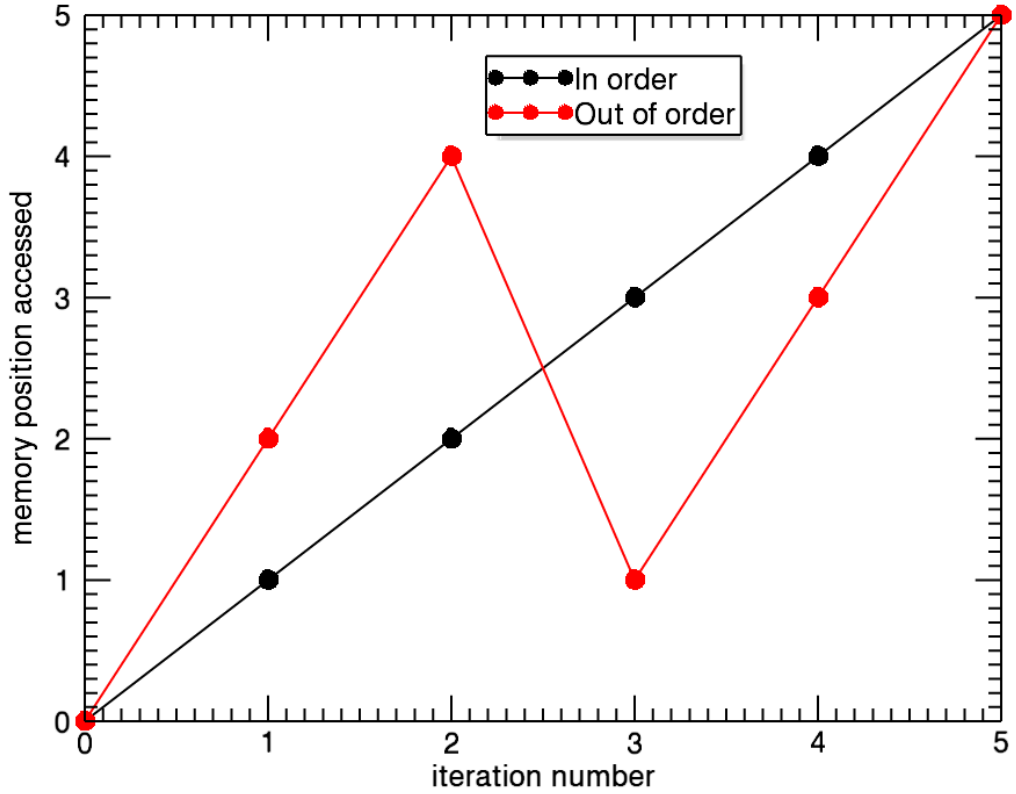
Arrays - armazenamento MD - exemplos

Já se os elementos são lidos fora da ordem (com os loops invertidos):

```
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

Há muitas idas e voltas (mostrados pela variável **k**, que indica que lugar na memória foi usado):

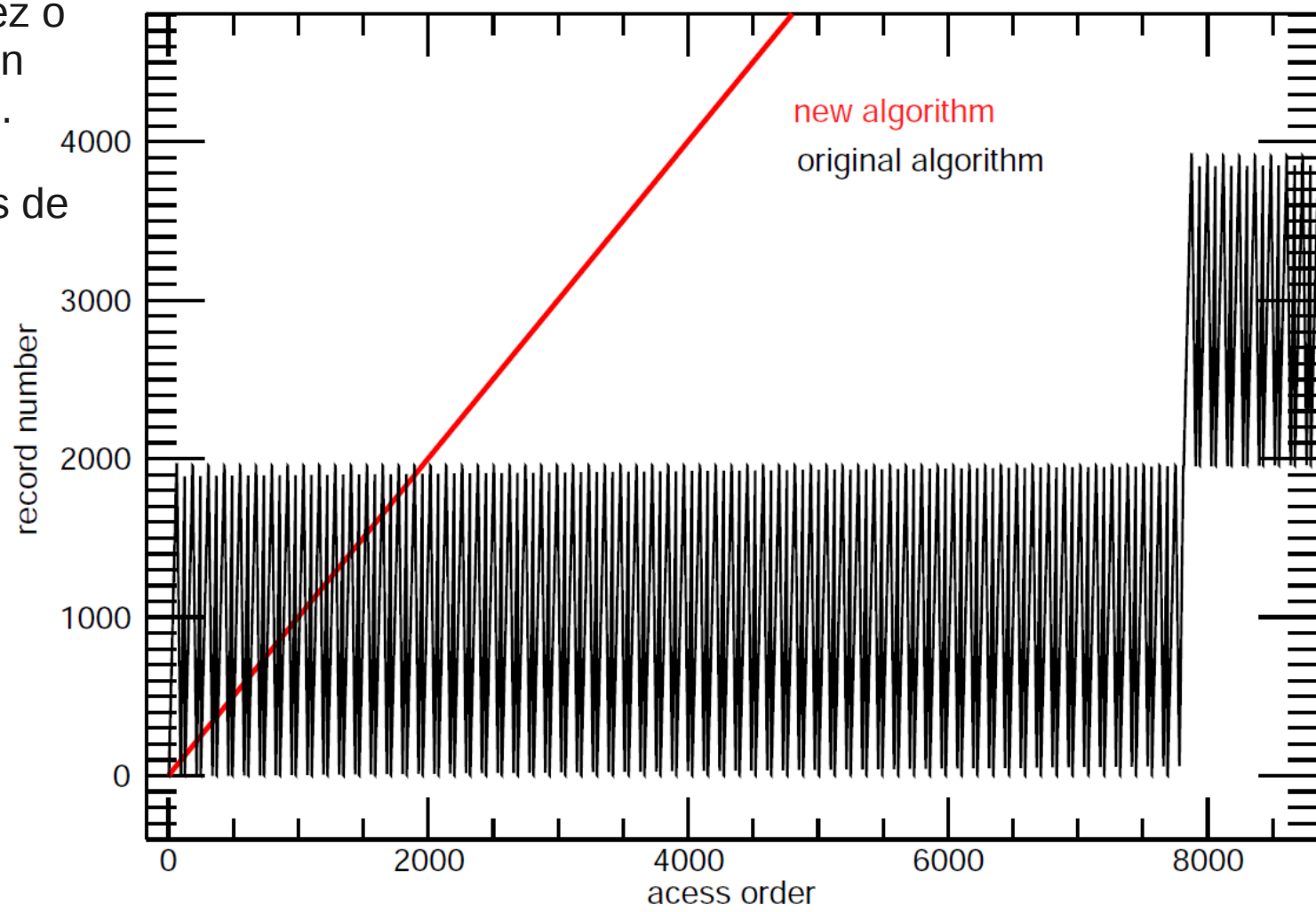


Arrays - armazenamento MD - exemplos

A diferença na ordem de acesso pode ser muito relevante para eficiência.

Neste exemplo real, passar a acessar pela ordem de armazenamento (no caso, em disco, onde é mais crítico) fez o programa que tomava 60 min passar a levar apenas 3 min.

E a mudança afetava menos de 10 linhas no código-fonte original.



Listas - definição

Elementos (*valores*) são armazenados **sequencialmente**, e acessados por seus índices

- **Semelhantes a arrays 1D.**

Ao contrário de arrays, são dinâmicas e, em algumas linguagens, heterogêneas (IDL, Python, R, Perl)*. Ex. (IDL):

```
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
```

Cria uma lista vazia

Elementos adicionados à lista

Remove o elemento da posição **1**.
Se a posição não é especificada, o último elemento é removido

Adiciona elemento, à posição 1. Quando a posição não é especificada, ocorre no final da lista.

*Contêiners em Java e C++ são sempre homogêneos; é possível misturar tipos fazendo contêiners de superclasses, mas isso gera algumas limitações aos objetos obtidos deles. Há melhores possibilidades com a biblioteca Boost (C++).

Listas - características

Eficientes para adição e remoção de elementos:

- Elementos (individuais ou seqüências de vários) podem ser adicionados ou removidos em qualquer posição, a qualquer momento.
- Na maioria das listas o default é adição / remoção do final.

Muito adequadas para:

- Quando não se sabe previamente quantos elementos vão ser acumulados.
- Quando não se sabe previamente os tipos / dimensões dos elementos.
- Quando há adições e remoções de elementos da lista, em particular para pilhas e filas, (discutidas adiante).

Listas - características

Suportam naturalmente montar arrays não uniformes, sem a necessidade de indireção por ponteiros. Exs. (IDL):

```
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
```

Ou, no lugar de:

```
something=(*(*state_pointer)[big_array_index])[first_col,*]
```

Pode ser usado simplesmente:

```
something=(state_list[big_array_index])[first_col,*]
```

Ou (com **pp_list**, ou **list** da versão 8.1):

```
something=state_pp_list[big_array_index,first_col,*]
```

Listas - exemplos

Os mesmos exemplos usados para ponteiros, para arrays não regulares (tanto observações como modelos):

- Elementos de
  - Famílias de asteróides
  - Aglomerados de estrelas / galáxias
  - Sistemas estelares múltiplos
  - Sistemas planetários
  - Organizações hierárquicas

- Vizinhos de
  - Asteróides (identificação de famílias, classificação de composições)
  - Estrelas (classificações por cores ou indicadores espectrais)
  - Galáxias (classificações por cores, indicadores espectrais, velocidades, forma)

- Linhas / bandas em espectros
  - Linhas / bandas de mesma origem (mesmo íon / elemento / molécula / partícula)
  - Vizinhos (identificação de linhas dominantes e contaminações)

- Observações / resultados de modelos
  - Repetidas observações do mesmo objeto.
  - Diferentes observações do mesmo objeto (diferentes instrumentos / modos).

- Grades não regulares
  - Parâmetros de modelos (modelos calculados para diferentes valores dos parâmetros).
  - Espaçamento não regular em grades espaciais.
  - Modelos com diferentes números de objetos / espécies em diferentes células.

Listas - exemplos

Armazenar vários arrays, onde cada um tem um número diferente pontos. Ex. (IDL):

```
files=file_search('*.fits',count=nfiles)
wavelengths=list()
fluxes=list()
for i=0,nfiles-1 do begin
  read_spectrum(files[i]),wavelength,flux
  wavelengths.add,wavelength
  fluxes.add,flux
endfor
```

Cada elemento de *wavelengths* e *fluxes* tem comprimentos de onda / fluxos de cada espectro:

```
IDL> help,wavelengths,fluxes
WAVELENGTHS     LIST  <ID=1  NELEMENTS=3>
FLUXES          LIST  <ID=2  NELEMENTS=3>
IDL> help,wavelengths[1],fluxes[1]
<Expression>    DOUBLE    = Array[2000]
<Expression>    DOUBLE    = Array[2000]
IDL> help,wavelengths[2],fluxes[2]
<Expression>    DOUBLE    = Array[1024]
<Expression>    DOUBLE    = Array[1024]
```

O que pode ser usado, por exemplo,

```
some_result=dblarr(nfiles)
for i=0,nfiles-1 do begin
  some_result[i]=some_function(wavelengths[i],fluxes[i])
  !null=plot(wavelengths[i],fluxes[i],name=some_result[i],/over)
endfor
```

Listas - exemplos

Alguns usos de arrays não regulares e adição / remoção dinâmica de elementos:

- Armazenar vários espectros de objetos / instrumentos diferentes, onde cada um tem um número diferente de pontos.
- Armazenar um número variável de espectros dos objetos (não foram observados o mesmo número de vezes).
- Armazenar quais são os vizinhos de objetos dentro de um limite (de distância geométrica, de distância em espaço de parâmetros, de distância entre linhas espectrais); o número de vizinhos de cada objeto é variável.

Listas - exemplos

Ex (IDL): Encontrar em uma lista de posições (ex: comprimentos de onda), grupos, que têm elementos dentro de um limite de distância.

```
IDL> restore,'objects_to_process.sav',/verbose
% RESTORE: Restored variable: INTENSITIES.
% RESTORE: Restored variable: POSITIONS.

IDL> help,intensities,positions
INTENSITIES     LIST  <ID=2597  NELEMENTS=1297>
POSITIONS       LIST  <ID=2  NELEMENTS=1297>

IDL> print,intensities[0:3]
      99.973708
      99.871224
      99.734586
      99.617141

IDL> print,positions[0:3]
      7135.9932
      1680.8704
      7813.9323
      387.11663
```

`intensities` foi usado apenas para ordenar as listas, começando da intensidade mais alta. Assim os grupos de vizinhos vão ser arranjados a partir do maior elemento de cada grupo.

Listas - exemplos

A lista é processada, identificando os vizinhos dentro de uma distância limite*:

```
group_leaders=list() ;list where the groups will be stored when found
while (positions) do begin ;loops as long as the list is not empty
  current_pos=positions[0]

  ;find out which objects are neighbours of the current one
  tmp=positions.toarray() ;make an array out of the list
  w=where(abs(tmp-current_pos) lt max_distance,nw,/null)
  neighbours_positions=list()
  neighbours_positions.add,tmp[w],/extract

  ;make the element to go in the list of processed elements
  group_leaders.add,{position:positions[0],intensity:intensities[0],$
    neighbours_positions:neighbours_positions}

  ;remove the used elements from the two lists
  positions.remove,[w,0] & intensities.remove,[w,0]
endwhile
```

Ao final:

- As listas de elementos não processados (*positions* e *intensities)* vão estar vazias.
- Na lista group_list há um elemento para cada grupo de vizinhos (dentro da distância max_distance), que contém a posição e intensidade do elemento de mais alta intensidade, e uma (potencialmente vazia) lista de seus vizinhos.

Listas - exemplos

Resultados exemplo:

max_distance=100:

```
IDL> help,group_leaders
GROUP_LEADERS   LIST  <ID=5192  NELEMENTS=69>

IDL> help,group_leaders[0]
   POSITION             DOUBLE           7135.9932
   INTENSITY            DOUBLE           99.973708
   NEIGHBOURS_POSITIONS OBJREF    <ObjHeapVar5193(LIST)>

IDL> help,(group_leaders[0]).neighbours_positions
<Expression>    LIST  <ID=5193  NELEMENTS=26>
```

max_distance=1000:

```
IDL> help,group_leaders
GROUP_LEADERS   LIST  <ID=5192  NELEMENTS=7>

IDL> help,group_leaders[0]
** Structure <e02e2a28>, 3 tags, length=24, data length=20, refs=2:
   POSITION             DOUBLE           7135.9932
   INTENSITY            DOUBLE           99.973708
   NEIGHBOURS_POSITIONS  OBJREF    <ObjHeapVar5193(LIST)>

IDL> help,(group_leaders[0]).neighbours_positions
<Expression>    LIST  <ID=5193  NELEMENTS=248>
```

Mapas - características

Semelhantes a estruturas: armazenam valores, por nomes (**chaves**).

Ao contrário de estruturas, **as chaves não precisam ser strings** (mas podem ser).

Ao contrário de índices (arrays e listas), as chaves **não são sequenciais**, nem precisam ser inteiras: um mapa pode ter um único elemento de chave **2543**, ou **73.24**.

Ao contrário de estruturas, mapas são dinâmicos: elementos podem ser adicionados e removidos à vontade (e é algo eficiente).

- Mapas estão para estruturas (aproximadamente) como listas estão para arrays 1D.

Em linguagens dinâmicas são heterogêneos: tipos das chaves e valores podem ser misturados.

Em linguagens dinâmicas os tipos e dimensões dos valores podem ser alterados à vontade.

Elementos não são armazenados em ordem*:

- A ordem em que chaves são listadas não é a mesma que a ordem em que foram colocadas.
- Não faz sentido seleções por fatias (ex: **`h['a':14.59]`**).
- Mas faz sentido seleções vetoriais (exs: **`h[[1,9,17,178]]`**, **`h[indgen(5)]`**).

Descobrir se uma chave está presente, e acessar o valor de uma chave, são **operações que levam um tempo constant**e: não importa se o mapa tem 10 elementos ou 1 milhão.

*Estritamente, ficam em uma ordem, mas não é uma ordenação óbvia.

Mapas – usos básicos (ex. IDL):

```
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
```

Cria um mapa vazio

Adiciona valores ao mapa

Mapas - exemplo – agrupamento de muitas variáveis

Como ler vários arquivos que contém 35 arrays cada, que dependem de 14 dimensões diferentes, com um só comando e sem criar 49 variáveis por arquivo?

```
$ ncdump -h refspec_g01_0.nc
netcdf refspec_g01_0 {
dimensions:
        nlay = 51 ;
        nwn = 400 ;
        nleg = 33 ;
        numu = 2 ;
        nlev = 52 ;
        nphi = 3 ;
        ngas = 2 ;
        nwnc = 1 ;
        scal = 1 ;
        v3 = 3 ;
        dnl = 1 ;
        nk = 1 ;
        tdisr = UNLIMITED ; // (0 currently)
        na = 16 ;
variables:
        float alb(nwn) ;
        float z(nlay) ;
        float t(nlay) ;
        float p(nlay) ;
        float wl(nwn) ;
        float iof(nwn) ;
        float mtau(nwn) ;
        float htau(nwn) ;
        float hctaus(nwn) ;
        float gtau(nwn) ;
        float outcos(scal) ;
        float phi(nleg) ;
        float phic(scal) ;
        float umu(numu) ;
        float flux(v3, nwn, nlev) ;
        float mix(nlay, ngas) ;
        float c(nlay) ;
        float psat(nlay) ;
        float ga(nwnc) ;
        float wlc(nwnc) ;
        float inc(scal) ;
        int dm(scal) ;
        int ord(scal) ;
        float fbeam(scal) ;
        float wn(nwn) ;
        float tautot(nwn, nlay) ;
        float htaus(nwn, nlay) ;
        float htaux(nwn, nlay) ;
        float taug(nwn, nlay) ;
        float phase(nwn, nlay, nleg) ;
        float ssa(nwn, nlay) ;
        float uu(nwn, nphi, nlev, numu) ;
        float u0u(nwn, nlev, numu) ;
        float tray(nwn, nlay) ;
        float gtauo(tdisr, nwn, nlay) ;
}
```

Conteúdo de um arquivo NetCDF (mostrado por conversão para CDL):

- Há 14 variáveis usadas como dimensões, para as 35 variáveis que são arrays.
- **`alb(nwn)`** indica que **`alb`** é um array de **`nwn`** elementos.
- **`nwn`** está armazenado como uma das dimensões, e é **400.**

(mais sobre NetCDF e CDL na aula sobre arquivos)

Mapas - exemplos

Os conteúdo do arquivo pode ser colocado em um mapa. Ex. (IDL):

```
IDL> print,h
var_dims: <ObjHeapVar229(HASH)>
vars: <ObjHeapVar152(HASH)>
dims: <ObjHeapVar118(HASH)>

IDL> print,h['vars']
T:       173.203      175.459      175.848      176.001      175.975      175.839 ...
Z:       431.869      418.858      406.067      393.403      380.464      367.608 ...
FLUX:       3.00660      1.45536     0.705180     0.341669     0.165532    0.0801887 ...
PHASE:       1.00000     0.779034     0.660858     0.542052     0.449983     0.375222 ...
TAUTOT:      0.694377     0.693420     0.693474     0.693540     0.693636     0.693739 ...
(...)

IDL> help,(h['vars'])['TAUTOT']
<Expression>    FLOAT     = Array[51, 400]

IDL> print,(h['var_dims'])['TAUTOT']
NLAY NWN

IDL> print,(h['dims'])[(h['var_dims'])['TAUTOT']]
NLAY:       51
NWN:      400
```

Ao contrário de estruturas, aqui campos podem ser adicionados e removidos, ou terem seus tipos ou dimensões alterados.

Mapas - exemplos

Agrupamento de variáveis relacionadas, onde cada elemento é uma coluna de tipo diferente de uma tabela. Ex. (IDL):

Do arquivo:

NAME	CALMPOS	FILNAME	ECHLPOS	DISPPOS	TARGNAME	POSDIR	CLASS	MJD-OBS	ITIME	COADDS
dec18s0001	0	NIRSPEC-5-AO	62.6300	36.4500	HD85258	NIRSPEC-5-AO/p1	STAR	54087.57421875	100.00000	1
dec18s0002	0	NIRSPEC-5-AO	62.6300	36.4500	HD85258	NIRSPEC-5-AO/p1	STAR	54087.57421875	100.00000	1
dec18s0003	1	NIRSPEC-5-AO	62.6300	36.4500	HD85258	NIRSPEC-5-AO/p1	FLAT	54087.57812500	4.60000	5
dec18s0004	1	NIRSPEC-5-AO	62.6300	36.4500	HD85258	NIRSPEC-5-AO/p1	DARK	54087.57812500	4.60000	5

Se obtém o hash:

```
IDL> help,h
H               HASH  <ID=44  NELEMENTS=11>
IDL> print,h
NAME: dec18s0001 dec18s0002 dec18s0003 dec18s0004
POSDIR: NIRSPEC-5-AO/p1 NIRSPEC-5-AO/p1 NIRSPEC-5-AO/p1 NIRSPEC-5-AO/p1
CLASS: STAR STAR FLAT DARK
COADDS:            1           1           5           5
ITIME:       100.000      100.000     4.60000     4.60000
DISPPOS:       36.4500      36.4500     36.4500     36.4500
MJD_OBS:       54087.6      54087.6     54087.6     54087.6
ECHLPOS:       62.6300      62.6300     62.6300     62.6300
FILNAME: NIRSPEC-5-AO NIRSPEC-5-AO NIRSPEC-5-AO NIRSPEC-5-AO
CALMPOS:            0           0           1           1
TARGNAME: HD85258 HD85258 HD85258 HD85258
```

(como fazer a leitura será discutido na aula sobre arquivos)

Mapas - exemplos

Manter a associação de informações relacionadas:

- Campos de dados, mantidos associados a campos de dados relacionados, e campos para indicar atributos dos dados (metadados). Ex (IDL): No lugar de inventar nomes, carregar, e manter sincronizadas várias variáveis:

```
files=file_search('./*.fits')
object_names=list()
wavelengths=list()
fluxes=list()
instrument_data=list()
(...)
for i=0,n_elements(files)-1 do begin
  read_spectrum,files[i],wavelength,flux,instrument,name
  object_names.add,name
  wavelengths.add,wavelength
  fluxes.add,flux
  instrument_data.add,instrument
  (...)
endfor
(..)
i=0
while (i lt n_elements(fluxes)) do $
 if do_not_keep(fluxes[i],wavelengths[i]) then begin
```

(a função do_not_keep() decide se o espectro dado deve ser descartado ou não)

```
  wavelengths.remove,i
  fluxes.remove,i
  object_names.remove,i
  instrument_data.remove,i
endif else i++
```

Mapas - exemplos

Com um mapa para cada observação, é muito mais simples manter a sincronia, e não é necessário carregar muitas variáveis separadas:

```
files=file_search('./*.fits')
objects=list()
for i=0,n_elements(files)-1 do begin
  read_spectrum,files[i],wavelength,flux,instrument,name

  objects.add,hash('name',name,'wavelength',wavelength,'flux',flux,$
    'instrument_data',instrument_data,...)

endfor
(..)
i=0

foreach element,objects,i do $
  if do_not_keep(element['wavelength'],element['flux']) then
objects.remove,i
```

Mapas - exemplos

Manter a associação de informações relacionadas em interfaces de rotinas: no lugar de passar muitos argumentos por palavras-chave, passa-se apenas um mapa. Ex. (IDL):

```
IDL>initial_parameters=hash()
IDL>initial_parameters['temperature']=5978d0
IDL>initial_parameters['resolution']=[200,100,50]
IDL>initial_parameters['name']='some_test_model_12b'
IDL>initial_parameters['composition']=read_composition('some_test_model_12b.nc')
IDL>initial_parameters['iterations']=500
```

O modelo é executado usando apenas uma variável (o hash com todos os parâmetros):

```
IDL>final_parameters=run_some_model(initial_parameters)
```

O hash retornado tem mais elementos, para os resultados, e os que não foram dados inicialmente (defaults foram usados):

```
IDL> print,final_parameters
temperature:        5978.0000
results: <ObjHeapVar135(HASH)>
wavelengths:        5379.3000       5379.3020       5379.3040       5379.3060...
composition: <ObjHeapVar112(HASH)>
resolution:      200     100      50
name: some_test_model_12b
iterations:      500
IDL> help,final_parameters['wavelengths']
<Expression>    DOUBLE    = Array[10000]
IDL> help,(final_parameters['results'])['OPTICAL_DEPTHS']
<Expression>    FLOAT     = Array[200, 100, 50]
```

Mapas - exemplos

Agora, muda-se apenas alguns parâmetros para executar o novo modelo:

```
IDL> final_parameters['wavelengths']=congrid(final_parameters['wavelengths'],30000)
```
(reamostragem dos 10000 comprimentos de onda, para 30000 pontos)

```
(...)
IDL> final_parameters['iterations']=10
IDL> final_parameters['resolution']=[400,200,100]
```
(mudança das dimensões da grade, o que não seria possível com estruturas)
```
(...)
IDL>new_parameters=run_some_model(final_parameters)
```

Agora o resultado tem as novas dimensões:

```
IDL> help,(new_parameters['results'])['OPTICAL_DEPTHS']
<Expression>    FLOAT     = Array[400, 200, 100]
```

Os parâmetros podem ser mudados localmente no código onde se calcula os novos valores. Não é necessário inventar muitas novas variáveis separadas para cada chamada do modelo:

```
new_wavelengths=congrid(wavelengths,30000)
(...)
new_iterations=10
new_resolution=[400,200,100]
(...)
run_some_model,wavelengths=new_wavelengths,temperature=temperature,$
  resolution=new_resolution,name=name,$
  composition=read_composition('some_test_model_12b.nc'),iterations=new_iterations,$
  results=new_results,...
```

Mapas - exemplos

Armazenamento de informações através de chaves, no lugar de fazer buscas em índices. Ex. (IDL): Armazenamento de vários espectros, pelo nome do objeto observado:

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

O que resultaria em

```
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]
```

Mapas - exemplos

Armazenamento de vários espectros, pelo nome do objeto observado:

Por que colocar em uma hash? Porque no código que vai usar os espectros, é muito mais cômodo. Ex. (IDL):

```
resample_data,spectra['HR5438']
!null=plot((spectra['HR5438']).wavelength,(spectra['HR5438']).flux)
```

que a alternativa com arrays e estruturas:

```
w=where(spectra.target eq 'HR5438')
resample_data,spectra[w],new_spectrum
!null=plot(new_spectrum.wavelength,new_spectrum.flux)
```

Onde ainda é necessário retornar o espectro reamostrado em outra variável (**`new_spectrum`**), porque não se pode mudar as dimensões de **`(spectra[w]).wavelength`** e **`(spectra[w]).flux`**.

Além de não poder colocar os espectros reamostrados de volta no array `spectra`, este array nem poderia ter sido construído se os elementos não fossem todos estruturas com as mesmas dimensões dos campos (por exemplo, porque alguns objetos foram observados em instrumentos diferentes).

Mapas - exemplos

Mais liberdade de escolha de chaves:

- Strings são arbitrárias, sem as limitações de caracteres de campos de estruturas, que não podem ter símbolos especiais como espaços em branco ou **`-+*/\()[]{} ,”'`**.

- Caracteres especiais comumente aparecem em chaves:
  - nomes de arquivos (**`some-file.fits`**)
  - nome do objeto (**`alpha centauri`**)
  - identificador em catálogo (**`PNG 004.9+04.9`**)
  - classificação do objeto (**`[WC6]`**), etc.

- Podem ser usadas chaves que não sejam strings:
  - doubles (data juliana)
  - inteiros não consecutivos e não começados em 0 (dia, número de catálogo, número de ordem do arquivo), etc.

Mapas - exemplos

Mais liberdade de escolha de chaves:

Ex. (IDL): Armazenamento de dados de linhas espectrais pelos comprimentos de onda:

```
lines=hash()
for i=0,nfiles-1 do begin
  read_element_file,files[i],element_data
  foreach el,element_data do lines[el.wavelength]=el
Endfor

IDL> print,lines
6398.9548: { argon 98.735900  0.0087539000  12.983800   3 NIST ...}
6835.7300: { argon 57.932800  0.0083250000  21.385000   2 NIST ...}
5293.3650: { neon 103.45920   0.0039636000  3.4285900   1 NIST ...}

IDL> help,lines[6398.9548d0]
** 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
```

Mapas - exemplos

No lugar de uma estrutura com ponteiros nos campos (para permitir trocar tipos/dimensões, trocando os alvos dos ponteiros).

Ex. (IDL): no lugar de:

```
something=(*(*state_pointer).big_array_pointer)[first_col,*]
```

É muito mais legível

```
something=(state_hash['big_array'])[first_col,*]
```

Ou (com **pp_hash**, ou **hash** da versão 8.1):

```
something=state_pp_hash['big_array',first_col,*]
```

Outros contêiners

Estruturas são normalmente implementadas como um tipos, e são a base para a definição de classes (por isso foram apresentadas na aula de tipos), mas são também contêiners – **heterogêneos, estáticos e não sequenciais**:

```
** 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
```

Mapas estão para estruturas (ambos não sequenciais) assim como listas estão para arrays (ambos sequenciais): **o segundo é a versão dinâmica do primeiro.**

Por isso **arrays, listas, estruturas e mapas são os contêiners mais fundamentais**: outros contêiners em geral são especializações / extensões / modificações destes 4.

Outros contêiners

Ponteiros e objetos costumam ser usados internamente na implementação de contêiners.

Python+Numpy, em particular, faz muito uso de ***views*** de contêiners:

- Internamente podem ser vistos como ponteiros para elementos do contêiner
- Com a aparência de novas variáveis: cria-se uma nova variável, mas que ainda usa a mesma memória dos elementos correspondentes no contêiner.

Objetos podem ser vistos como contêiners, já que podem armazenar dados.

- Com herança de classes é fácil criar a partir de contêiners existentes outros com funções ou propriedades adicionais, ou especializações dos existentes.

- Objetos são entidades ativas, que realizam operações sobre os dados que contém (inclusive armazenar dados dentro deles, e fornecer dados armazenados). Normalmente, externamente só se vê estas operações (metódos): só se vê a funcionalidade, e não o que acontece dentro deles.

Ponteiros são apenas referências, usados internamente ou explicitamente em quase tudo

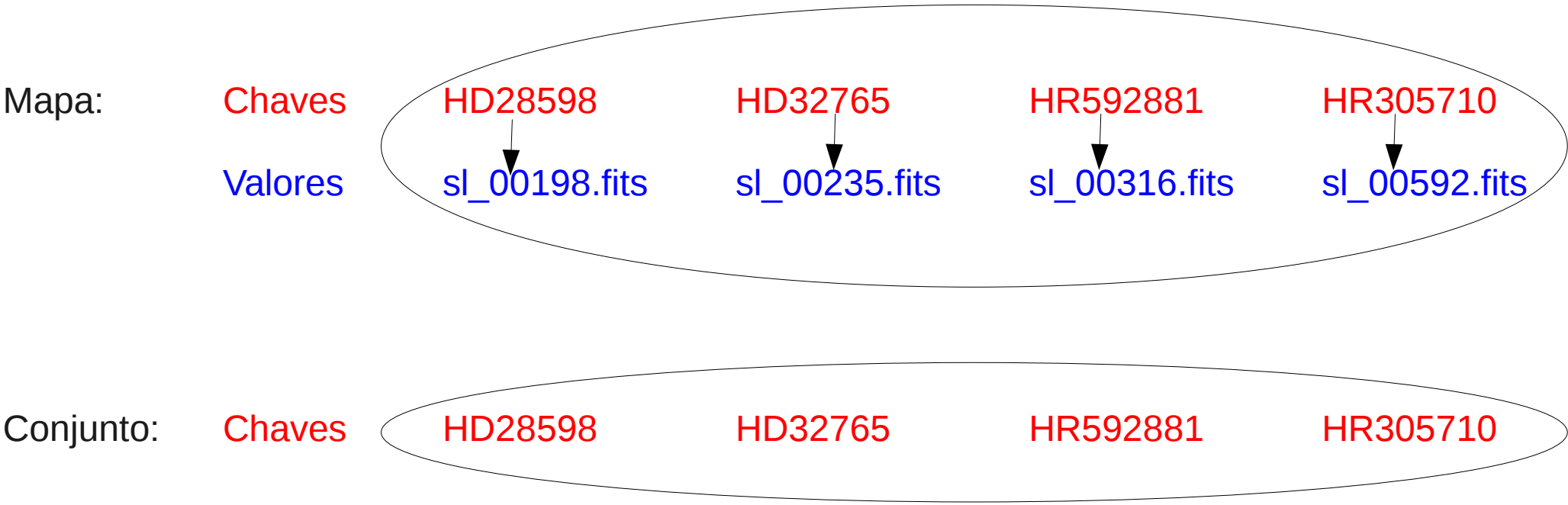
- Seu uso explícito tem menos utilidade nas linguagens mais modernas (ou modernizadas), tendendo a desaparecer, substituído por abstrações de mais alto nível e de uso mais direto.

Outros contêiners

Outros contêiners mais comuns:

Conjuntos (*sets*) - semelhantes a mapas, mas só armazenando chaves, sem valores associados a elas. Como conjuntos em matemática:

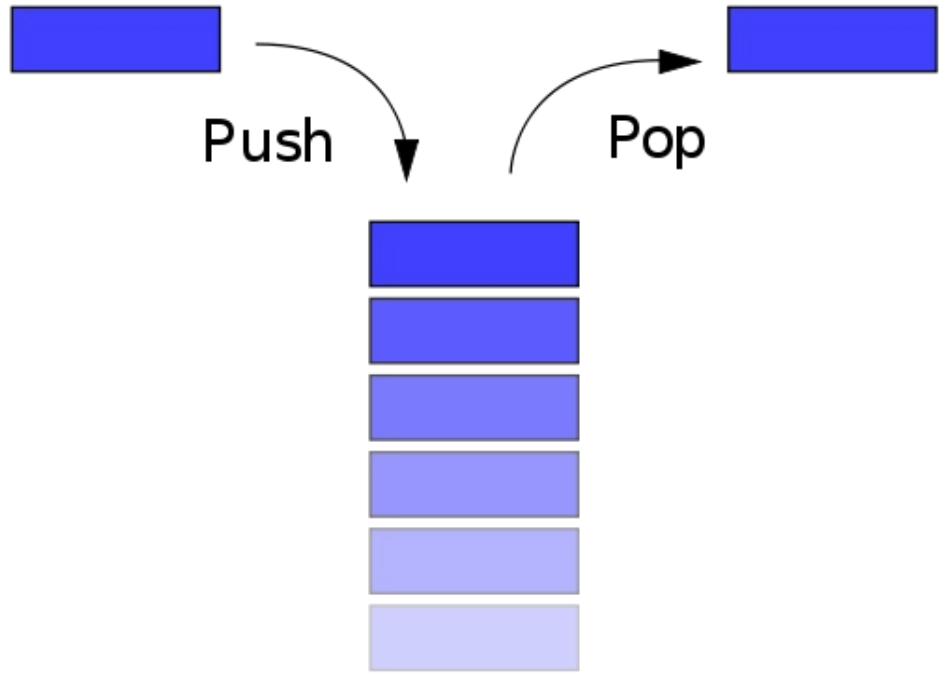
- Usos comuns: listas de elementos sem repetições - ir adicionando objetos a uma lista sem ter que verificar se já estão presentes, remover elementos duplicados.
  - ➔ Ex: listas de objetos observados, listas de arquivos usados (como listas de arquivos de calibração usados no processamento de um conjunto de observações), listas de datas de observações, listas de modelos utilizados dentre um grande conjunto, etc.
- Importantes pela utilidade de operações de grupo: *união, interseção, diferença*.
- Mapas podem ser usados no seu lugar, usando-se as chaves e ignorando os valores.



Outros contêiners

Pilhas (*stacks*) – Listas onde elementos só são adicionados e removidos ao final:

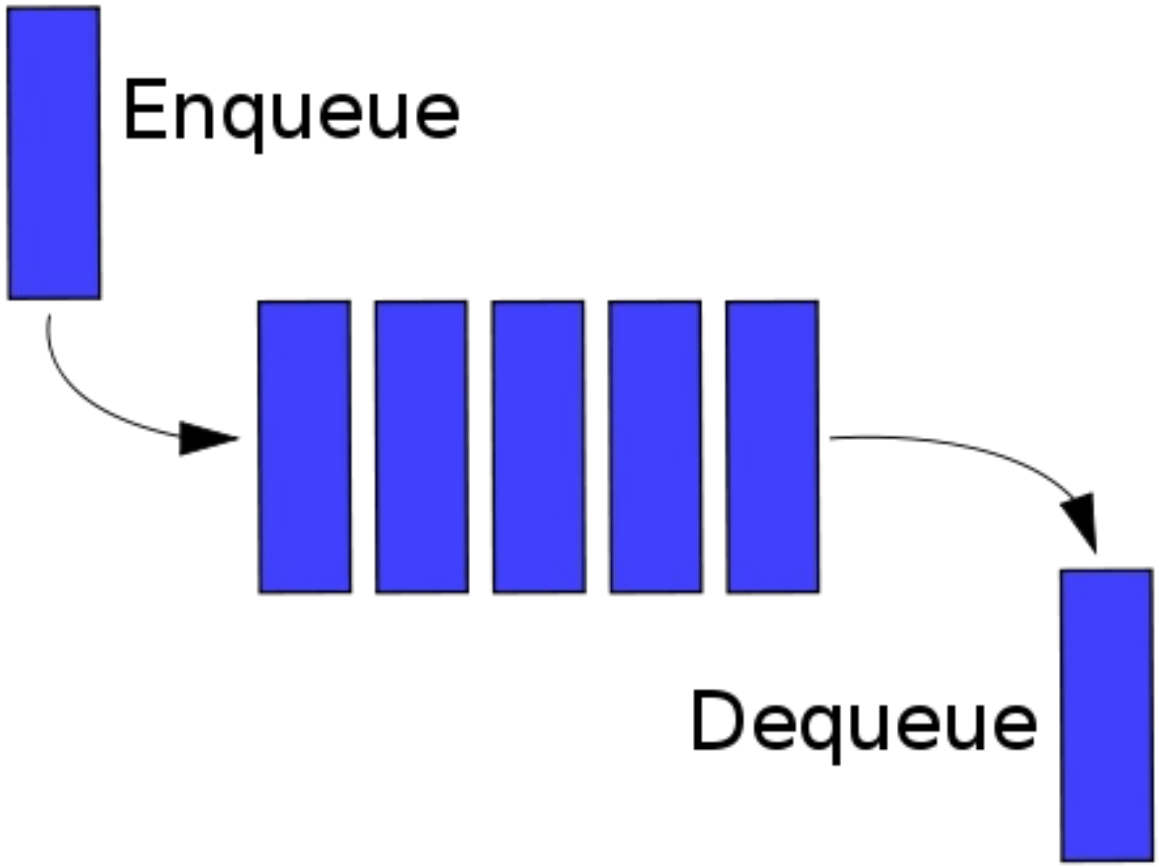
- Como uma pilha física: não dá para tirar o livro de baixo da pilha, nem adicionar um no meio da pilha; só em cima dá para colocar e tirar livros.
- Contêiners deste tipo são chamados de ***LIFO – Last In, First Out***.
- Úteis principalmente em programação geral: *call stacks* (a pilha de rotinas chamadas até um ponto: chamar uma nova rotina a adiciona à pilha, sair dela a remove da pilha), e *parsers* (interpretadores léxicos, especialmente em sistemas RPN (*Reverse Polish Notation*; exs: Postscript e BibTeX).
- Comuns em programação paralela, onde os trabalhos a executar são colocados na fila pelo processo que controla o trabalho a fazer, e retirados da fila pelos processos que executam os trabalhos.



Outros contêiners

Filas (*queues*) – Listas **FIFO (*First In, First Out*)**: elementos só são adicionados ao final, e só removidos do começo.

- Uso mais comum em programação geral, em particular para os mesmos usos de gerenciamento de trabalho entre processos mencionados para pilhas (a ordem de execução sendo inversa à das pilhas).

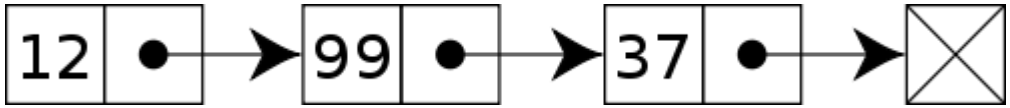


Outros contêiners

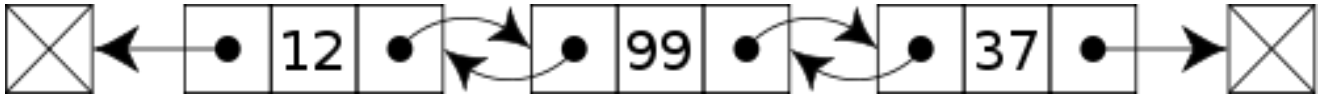
Linked Lists - são um conceito mais primitivo que as listas mostradas aqui (listas indexadas):

- São listas, mas sem que haja a possibilidade de acesso aleatório (acesso a elementos por índices).
- Todos os acessos têm que começar por um elemento, e só se pode navegar para o elemento anterior e/ou posterior ao atual: acessos são necessariamente em seqüência.
- Em algumas implementações, a lista pode nem saber quantos elementos tem (é necessário ir do primeiro ao último, contando, para o saber).
- Tem os mesmos usos que listas indexadas, mas por estas limitações, tendem a ser menos usados.
- Em algumas implementações, podem ser mais eficientes que listas indexadas **para acesso sequencial**.
- São muitas vezes usados internamente na implementação de listas de mais alto nível (como listas indexadas).

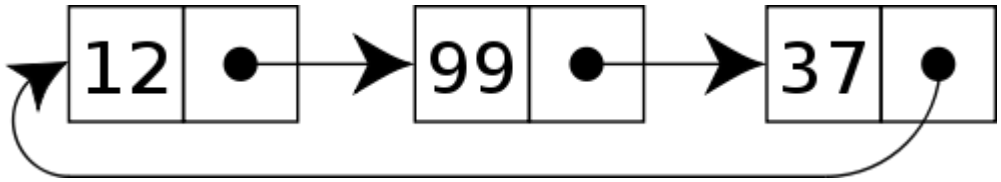
Unidirecional



Bidirecional



Cíclica (“circular”)



Outros contêiners - árvores

Árvores - contêiners não sequenciais onde o acesso também não ocorre por nomes: usa-se uma estrutura hierárquica:

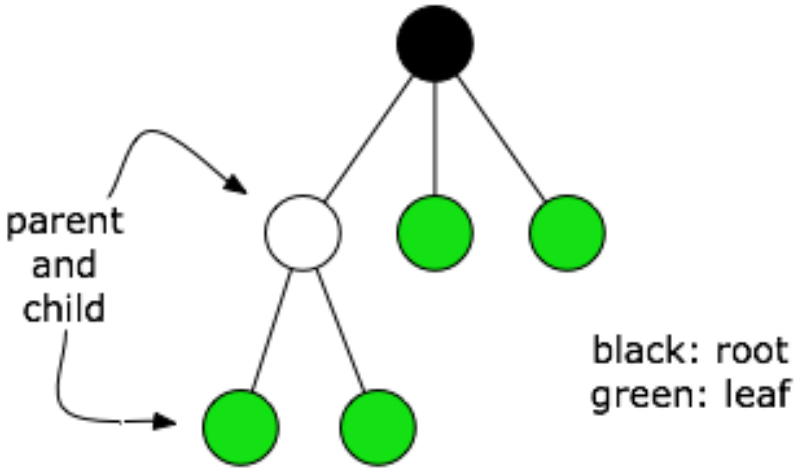
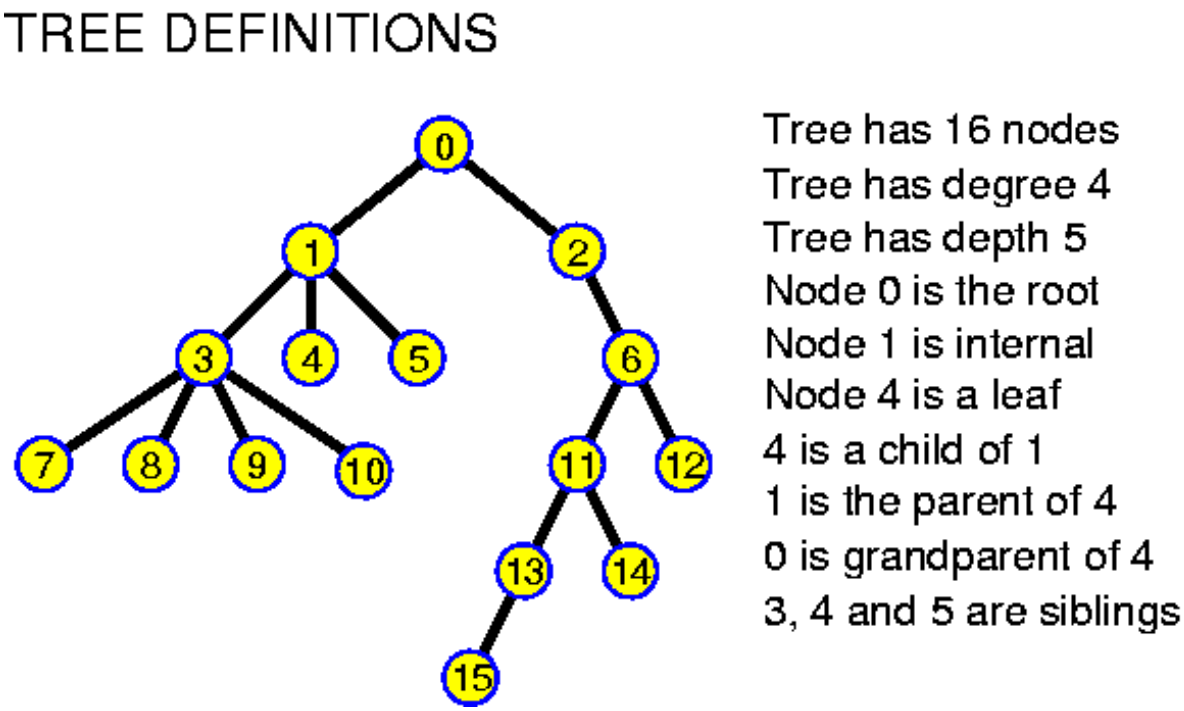


Figure: tree data structure



Cada nó tem zero ou mais filhos (ramos descendentes) e apenas zero ou um pai (ramo ascendente)

- Apenas o nó raiz não tem pai, todos os outros são filhos de alguém.
- Os nós sem filhos também são chamados de folhas.

Navegação ocorre apenas verticalmente: de filhos para pais ou pais para filhos, sempre a partir de algum nó.

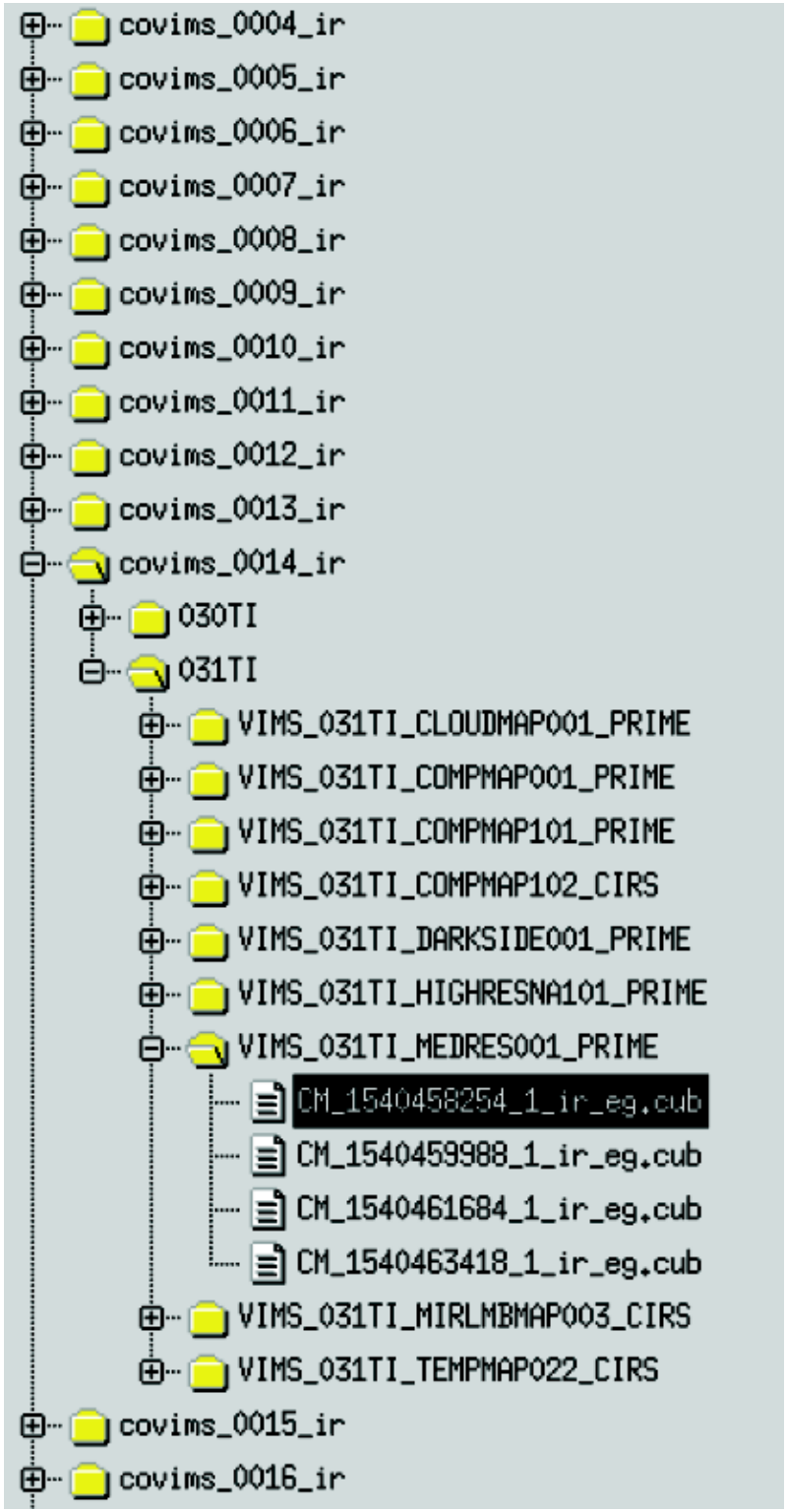
Em árvores binárias só há dois filhos por nó - usadas especialmente onde cada nó representa uma pergunta, que só tem resposta sim ou não.

Outros contêiners - árvores

Ex:

- Árvore usada para organizar arquivos de observações em um banco de dados, mostrada em uma interface gráfica.
- Esta é uma estrutura de árvore, mas não é um conjunto de diretórios, como pode parecer.

  Estruturas de diretórios são o uso mais comum de árvores.

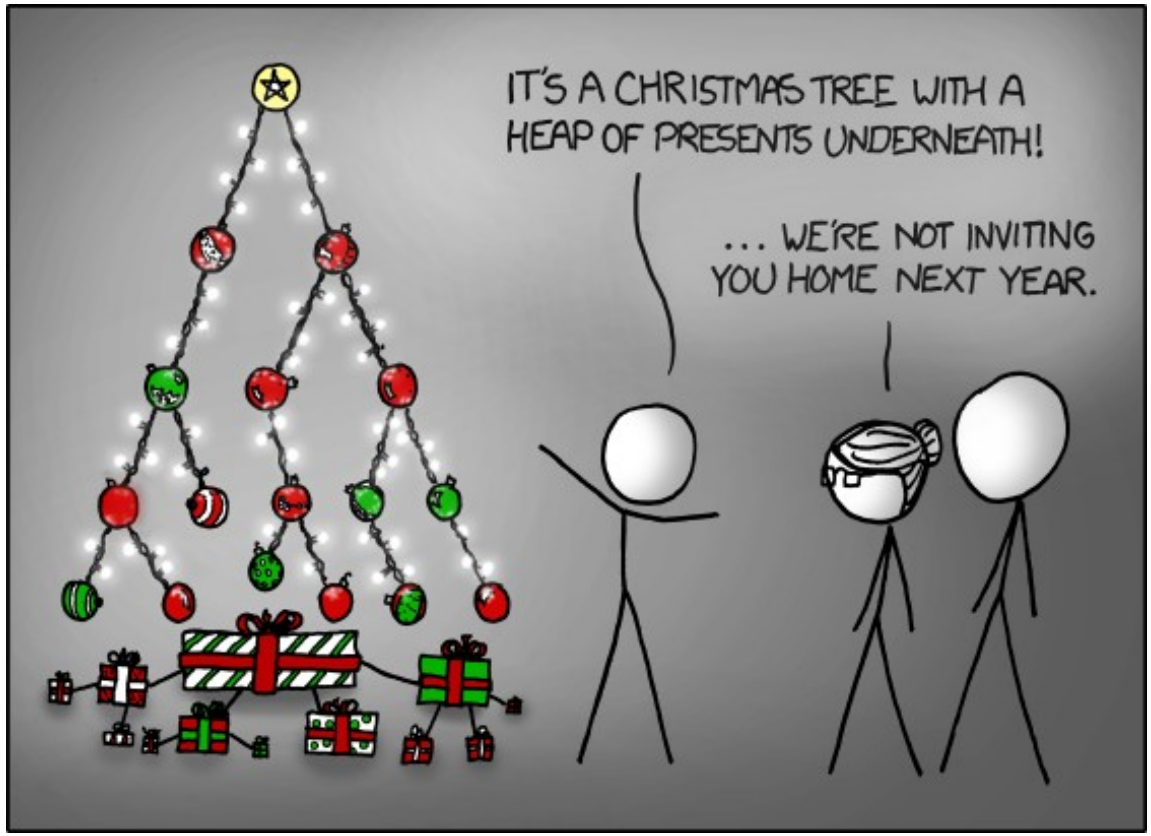


Outros contêiners - árvores

***Heaps* (filas prioritárias)** são outra especialização de árvores, onde nós têm chaves, e filhos sempre têm chaves de valor menor ou igual ao das chaves de seus pais.

Árvores são usadas principalmente para:

- Organizar dados hierárquicos, de forma semelhante a diretórios: observações hierarquizadas (missão -> data -> instrumento -> objeto), modelos hierarquizados (algorítimo -> implementação -> ambientes/especializações -> conjuntos de parâmetros), etc.
- Processamento de arquivos hierárquicos, especialmente XML e HDF5 (aula adiante).
- Internamente em interfaces gráficas e visualizações gráficas, onde um pai é “dono” de seus filhos e ações sobre pais são propagadas a seus filhos.



(http://www.xkcd.org/835)

Vetorização - motivação

Computação científica costuma ser fortemente dependente de processar muitos elementos de contêiners – arrays em particular.

Em tempos arcaicos, era necessário o programador escrever explicitamente as operações em termos de cada elemento. Ex. (IDL):

```
for k=0,n1-1 do begin
  for  j=0,n2-1 do begin
    for i=0,n3-1 do begin
      c[i,j,k]=a[i,j,k]+b[i,j,k]
      d[i,j,k]=sin(c[i,j,k])
    endfor
  endfor
endfor
```

O que é ruim por muitos motivos:

- É muito trabalho para escrever o que conceitualmente é apenas **c=a+b, d=sin(c)**.
- Há uma grande possibilidade de erros: erros de digitação, erros no uso dos índices: É **c[i,j,k]** ou **c[k,j,i]**, ou **c[j,k,i]**? Quais são os limites das dimensões? Quais são os índices das dimensões?
- É uma execução serial (um elemento por vez) de uma operação que poderia ser (automaticamente) paralelizada.
- **Em algumas linguagens (dinâmicas, em particular) é extremamente ineficiente**.

Este ainda é um exemplo extremamente simples; em operações mais complicadas (adiante), a possibilidade de erro e a verbosidade desnecessária são muito mais relevantes.

Vetorização - motivação

Qual é a alternativa a escrever todos estes loops em contêiners?

Vetorização - expressões são escritas como a operação pretendida entre as entidades (os contêiners), da mesma forma que se faz em linguagem verbal ou matemática:

c=**a**+**b**
d=sin(**c**)

Onde **a** e **b** são arrays de qualquer dimensão (**a** e **b** têm as mesmas dimensões)

$\mathbf{x} = A \cdot y = U^T U = U \Sigma V^T \cdot \mathbf{y}$

$\mathbf{F} = q\,(\mathbf{E} + \mathbf{v} \times \mathbf{B})$

É o trabalho do compilador / interpretador realizar as operações sobre os elementos, mantendo a contabilidade dos índices.

Em geral, qualquer tarefa onde seja necessário manter a contabilidade de muitas coisas (como índices e dimensões) é adequada para computadores, não para pessoas.

Porque computadores têm memória perfeita (nunca se esquecem ou confundem) e muito grande, e podem iterar muito rápido sobre muitas coisas.

Vetorização - motivação

O programador só informa operações de mais alto nível - **operações vetoriais**. Ex. (IDL):

```
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]
```

Quando compiladores / interpretadores encontram operações assim, eles sabem qual é a idéia pretendida: sabem sobre que elementos têm que operar de que forma.

O software pode automaticamente paralelizar a execução.

É o mesmo que pessoas fariam, na mão: já se sabe que se trata de fazer o mesmo para cada elemento, não é necessário depois de cada elemento decidir o que fazer; se há várias pessoas, cada uma faz um pedaço.

Vetorização - possibilidades

O nível de vetorização suportado varia drasticamente entre linguagens. Da mais ingênua para a mais capaz:

- C, Fortran < 90 e Perl nada têm.
- Fortran 90, 95, 2003 e 2008 (em graus progressivos), C++, Java: vetorização simples:
  - desajeitada para mais de 1D (pior ainda para mais de 2D), limitada a operações sobre arrays inteiros ou fatias regulares (mostrada adiante)
- Biblioteca **Boost** (não padrão) para C++ provê boa funcionalidade para linguagens estáticas. Partes dela devem ser gradualmente incluídas nos próximos padrões.
- R tem melhor suporte a operações não triviais, especialmente até 2D (classe matrix).
- **Só IDL e Python+Numpy*** têm as operações de mais alto nível (especialmente para mais de 1D, inclusive com números arbitrários de dimensões), que dão muito mais poder e conveniência, eliminando muitos loops.

Algumas funcionalidades mostradas adiante só existem no nível de IDL e Python+Numpy.

*Numpy não é (ainda) parte das bibliotecas padrão de Python. Numpy é a biblioteca vetorial mais avançada atualmente, mas Python sem Numpy está em um nível entre o do C++ / Java e o nível do R.

Vetorização avançada (mas essencial) - Índices 1D x MD

Quando um array tem mais de 1D (MD), elementos podem ser selecionados pelos M índices. Ex. (IDL):

```
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
```

Ou por apenas um índice, que é a posição do elemento na ordem interna de armazenamento*:

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

O que é de utilidade comum, principalmente em funções que buscam elementos (mostradas adiante), que retornam índices 1D. A conversão MD->1D é mais simples:

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

Costuma haver funções prontas para fazer estas conversões.

*Essencial saber se o array é *row major* ou *column major*.

Vetorização avançada (mas essencial) - *fancy indexing*

Seleções de elementos por expressões vetoriais (***fancy indexing***, em Numpy):

- Quando se usa apenas um array de índices para outro array, o resultado tem a mesma dimensão que o array de índices (índices 1D), com os elementos correspondentes a cada posição no array de índices:

```
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
```

→ Array 1D, 4 elementos: 0,1,3,5 de a

→ Array 2D de índices (1D), 4 elementos

→ Array 2D, 4 elementos de a, dados pelos índices (1D) acima.

Da mesma forma poderia se usar como índices um array 3D, 4D, etc, que geraria um resultado das mesmas dimensões do array de índices: não importa a dimensão de `a`.

•Quando cada dimensão recebe um array de índices, os arrays de índices têm que ter dimensões iguais. O resultado tem estas dimensões, com os elementos selecionados pelo índice de cada dimensão na posição correspondente:

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

→ Array 1D, 2 elementos:
0: `a[0,3]`
1: `a[1,5]`

Vetorização avançada (mas essencial)

Operandos de dimensões diferentes: Operações vetoriais não se limitam a aplicar operações elemento-a-elemento entre arrays de mesma forma:

Operações com formas diferentes ocorrem por regras de conformidade: operações são permitidas entre arrays de dimensões **conformes** (em Numpy, dimensões que podem ser *broadcast* para se tornar iguais):

- Escalares sempre são confomes a qualquer array: o escalar é aplicado elemento a elemento.

- Se os arrays não têm dimensões iguais, o tratamento varia:
  - **IDL**: é usado o menor comprimento de cada dimensão, ignorando (truncado) o que sobra nos arrays onde as dimensões são maiores:

```
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
```

Vetorização avançada (mas essencial)

Operandos de dimensões diferentes: Operações vetoriais não se limitam a aplicar operações elemento-a-elemento entre arrays de mesma forma:

Numpy (Python): se em uma dimensão um array tem dimensão 1, este array tem seus elementos repetidos naquela dimensão até se tornar da mesma dimensão que o outro array.

```
>>> a = array([[1,2,3],[4,5,6]])
>>> b=array([[1],[2]])
>>> print(a)
[[1 2 3]
 [4 5 6]]
>>> print(b)
[[1]
 [2]]
>>> print(a+b)
[[2 3 4]
 [6 7 8]]
```

Ou seja, em IDL o resultado tem em cada dimensão o tamanho mínimo entre os operandos; em Numpy, o tamanho é o máximo entre os operandos (os de dimensão menor são repetidos até as dimensões ficarem iguais).

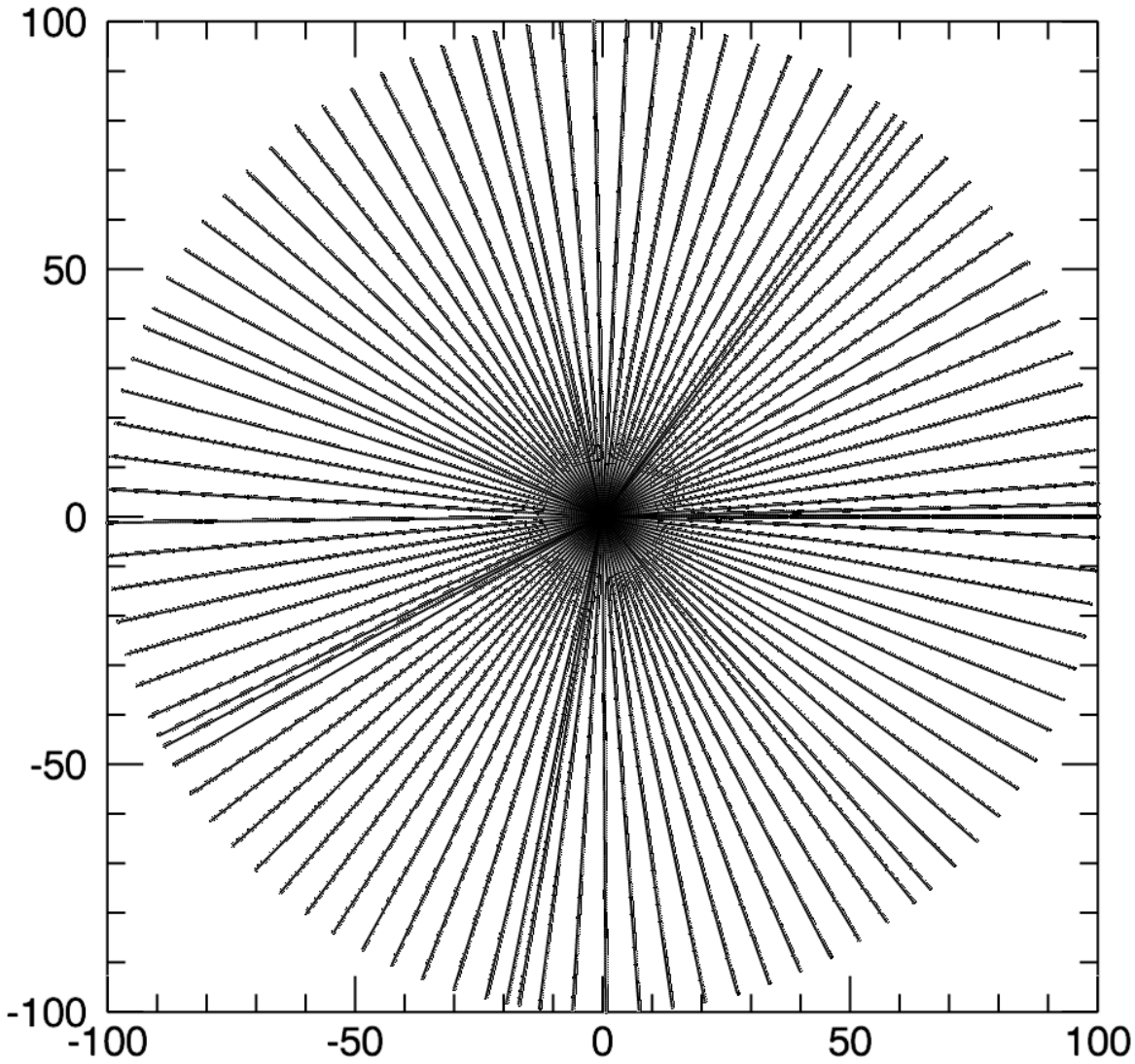
R, os elementos do array menor são repetidos o suficiente, ciclicamente:

```
> x=c(1,2,3)+c(10,20)
Warning message:
In c(1, 2, 3) + c(10, 20) :
  longer object length is not a multiple of shorter object length
> x
[1] 11 22 13
```

Vetorização avançada (mas essencial) - redimensionamento

É comum ter que mudar as dimensões de um array:

•Para conformidade com outros arrays. Ex: converter as coordenadas em uma grade polar (**r, theta**) para cartesianas, para obter os valores de uma função (**temperature**) em coordenadas cartesianas:



Vetorização avançada (mas essencial) - redimensionamento

Ex. (IDL):

```
IDL> help,temperature,r,theta,nr,ntheta
TEMPERATURE     FLOAT     = Array[200, 100]
R               DOUBLE    = Array[200]
THETA           DOUBLE    = Array[100]
NR              LONG      =          200
NTHETA          LONG      =          100

IDL> r_2d=rebin(r,nr,ntheta)
IDL> theta_2d=rebin(reform(theta,1,ntheta),nr,ntheta)
IDL> print,r_2d[0:3,0:2]
      0.10000000      0.60251256       1.1050251       1.6075377
      0.10000000      0.60251256       1.1050251       1.6075377
      0.10000000      0.60251256       1.1050251       1.6075377

IDL> print,theta_2d[0:3,0:2]
       0.0000000       0.0000000       0.0000000       0.0000000
       3.6000000       3.6000000       3.6000000       3.6000000
       7.2000000       7.2000000       7.2000000       7.2000000

IDL> x=r_2d*cos(theta_2d*!dpi/18d1)
IDL> y=r_2d*sin(theta_2d*!dpi/18d1)
IDL> help,r_2d,theta_2d,x,y
R_2D            DOUBLE    = Array[200, 100]
THETA_2D        DOUBLE    = Array[200, 100]
X               DOUBLE    = Array[200, 100]
Y               DOUBLE    = Array[200, 100]
```

Cada coluna corresponde a um raio, e cada linha a um ângulo, da mesma forma que **`temperature`**; a mesma relação será gerada para **`x`** e **`y`**, calculados para cada ponto do array **`temperature`**.

Vetorização avançada (mas essencial) - redimensionamento

•Para reduções. Ex: dadas n (4) coordenadas de pontos em 2D (cada linha tem x e y de cada ponto). Ex. (IDL):

```
IDL> xy=[[3d0,4d0],[6d0,8d0],[9d0,12d0],[12d0,16d0]]
IDL> print,xy
       3.0000000       4.0000000
       6.0000000       8.0000000
       9.0000000       12.000000
       12.000000       16.000000
```

Cada linha tem as coordenadas de um ponto

De onde se quer calcular a distância à origem de cada ponto:

Dimensão sobre a qual é realizada a soma.

```
IDL> r=sqrt(total(xy^2,1))
IDL> print,r
       5.0000000       10.000000       15.000000       20.000000
```

O mesmo algorítimo funcionaria, idêntico, se os pontos fossem em 3D:

```
IDL> print,xyz
       0.0000000       1.0000000       2.0000000
       3.0000000       4.0000000       5.0000000
       6.0000000       7.0000000       8.0000000
       9.0000000       10.000000       11.000000
IDL> r=sqrt(total(xyz^2,1))
IDL> print,r
       2.2360680       7.0710678       12.206556       17.378147
IDL> print,r^2
       5.0000000       50.000000       149.00000       302.00000
```

Vetorização avançada (mas essencial) - redimensionamento

Outras transformações (comuns para armazenamento / processamento):

Ex. (IDL): Uma rotina vetorial que processa n pontos em 3D, como arrays `[3,n]`. Mas os n pontos são coordenadas calculadas para cada posição em uma imagem, colocados em um array 3D. Ex. (IDL):

```
IDL> help,xyz
XYZ             FLOAT     = Array[3, 1000, 2000]
```

É necessário converter **`xyz`** de **`[3,ns,nl]`** para **`[3,ns*nl]`**, para poder usar na rotina:

```
IDL> sz=size(xyz,/dimensions)
IDL> print,sz
           3        1000        2000
IDL> xyz_2d=reform(xyz,sz[0],sz[1]*sz[2])
IDL> help,xyz_2d
XYZ_2D          FLOAT     = Array[3, 2000000]
IDL> r_theta_phi_2d=convert_to_spherical(xyz_2d)
```

O resultado tem a mesma forma de xyz_2d:

```
IDL> help,r_theta_phi_2d
R_THETA_PHI_2D  FLOAT     = Array[3, 2000000]
```

É necessário converter de volta a 3D:

```
IDL> r_theta_phi=reform(r_theta_phi_2d,sz[0],sz[1],sz[2])
IDL> help,r_theta_phi
R_THETA_PHI     FLOAT     = Array[3, 1000, 2000]
```

Vetorização avançada (mas essencial) - buscas

Buscas: **encontrar em um array elementos pelas suas propriedades** é uma das operações mais comuns. O que é muito facilitado por semântica vetorial. Ex. (IDL):

•Filtros:

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

(seleciona apenas os elementos de **`spectrum`** onde o campo **`wavelength`** tem valores entre 4d3 e 6d3)

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

(seleciona apenas os elementos de **`spectrum`** onde o campo **`flux`** não é **NaN** ou **infinito**)

•Elementos específicos

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

Se este tipo de operação tem que ser feito com vários ou todos os nomes de objetos, pode ser mais apropriado armazenar os objetos em um hash (onde o acesso é por nome), do que em um array, como acima:

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

Vetorização avançada (mas essencial) - buscas

Buscas: **encontrar em um array elementos pelas suas propriedades** é uma das operações mais comuns. O que é muito facilitado por semântica vetorial. Ex. (IDL):

•Elementos mais próximos de um valor:

- Muito necessário para encontrar elementos com valores reais (não inteiros), já que pode não haver elementos de valores exatamente iguais. Ex: encontrar qual elemento do array se trata da linha Hα:

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

Índice do elemento onde ocorre o mínimo

O mínimo procurado é dos módulos

•Local em uma seqüência monotônica que contém um valor.

- Ex: Em um modelo, alterar a temperatura nas células da grade que contém um certo raio (**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
```

Retorna o índice onde `r` (array ordenado) envolve o valor `r_search` (no caso, escalar, mas se fossem procurados vários, poderia ser um array).

Vetorização avançada (mas essencial) - inversão de índices

Com freqüência é necessário determinar quais são os índices onde cada valor ocorre em um array.

Necessário para ordenar / classificar / selecionar vários elementos de um array pelos seus valores.:

- Dados vários arquivos de observações:
  - Identificar quais foram feitas em cada data onde houve observações - ordem por inteiros (datas julianas) não necessariamente contíguos (possivelmente com intervalos de muitas unidades), não iniciados em 0.

  ➔ Identificar quais observações são de cada objeto / instrumento - ordem por strings.

- Dados vários modelos, identificar quais têm um dos parâmetros em cada caixa (faixa de valores).

É o mesmo que realizar uma busca para cada valor diferente de um array.

Mas fazer uma busca para cada valor é ineficiente, pois se está varrendo o array N vezes (havendo N valores diferentes).

Uma busca combinada só varre o array uma vez, e vai mantendo um registro do que encontra. É muito mais eficiente, e mais fácil de fazer.

Vetorização avançada (mas essencial) - inversão de índices

Ex. (IDL): encontrar quais observações foram feitas em cada data do conjunto:

```
IDL> help,obs
OBS             STRUCT    = -> <Anonymous> Array[10]

IDL> print,obs[0]
{ HD28985     2455563 sl_0092.fits<ObjHeapVar4(HASH)>}

IDL> help,obs[0]
** Structure <e41dc4b8>, 4 tags, length=48, data length=40, refs=3:
   OBJECT          STRING    'HD28985'
   DATE            LONG           2455563
   FILE            STRING    'sl_0092.fits'
   DATA            OBJREF    <ObjHeapVar4(HASH)>

IDL> print,obs.object
HD28985 HD59382 HD63281 HD63281 HD48561 HD78325 HR892561 HR748267
HR189365 HR167382

IDL> print,obs.date
     2455563     2455569     2455569     2455570     2455570
2455570     2455570     2455570     2455574     2455576
```

Vetorização avançada (mas essencial) - inversão de índices

Inversão é melhor realizada por algorítimos de histograma: a cada elemento do array, se verifica a que bin ele pertence, e se anota naquele bin o índice daquele elemento.

```
IDL>
h=histogram_pp(obs.date,min=min(obs.date),binsize=1L,reverse_list=rl,reverse_h
ash=rh,locations=locations)
```

Cada bin tem o número de observações de cada data:

```
IDL> foreach element,locations,i do print,element,' : ',h[i]
     2455563 :            1
     2455564 :            0
     2455565 :            0
     2455566 :            0
     2455567 :            0
     2455568 :            0
     2455569 :            2
     2455570 :            5
     2455571 :            0
     2455572 :            0
     2455573 :            0
     2455574 :            1
     2455575 :            0
     2455576 :            1
```

*__histogram_pp()__ é uma interface para a função __histogram()__ da biblioteca padrão, para fornecer os índices reversos em listas e/ou hashes, no lugar dos complicados arrays de __histogram()__. Pode ser encontrada em http://ppenteado.net/idl/pp_lib/doc/index.html

Vetorização avançada (mas essencial) - inversão de índices

Mas pouco interessa saber **quantas** são as observações de cada data. O objetivo é saber **quais** são de cada data. Para isso servem os *índices reversos* gerados na criação do histograma. Em forma de lista:

```
IDL> help,rl
RL              LIST  <ID=44  NELEMENTS=14>
IDL> foreach element,locations,i do print,element,' : ',rl[i]
   0      2455563 :            0
   1      2455564 : !NULL
   2      2455565 : !NULL
   3      2455566 : !NULL
   4      2455567 : !NULL
   5      2455568 : !NULL
   6      2455569 :            1           2
   7      2455570 :            3           4           5           6           7
   8      2455571 : !NULL
   9      2455572 : !NULL
  10      2455573 : !NULL
  11      2455574 :            8
  12      2455575 : !NULL
  13      2455576 :            9
```

Cada elemento da lista **`rl`** é um (possivelmente vazio) array, que tem os índices dos elementos (no caso, do array **`obs.date`**) que caem no bin correspondente. Ex: na data **2455569** houve duas observações, as de índices **1** e **2** no array **`obs`**:

```
IDL> print,locations[6],obs[rl[6]]
     2455569
{ HD59382     2455569 sl_0102.fits<ObjHeapVar8(HASH)>}
{ HD63281     2455569 sl_0110.fits<ObjHeapVar12(HASH)>}
```

Vetorização avançada (mas essencial) - inversão de índices

Para alguns usos, pode ser mais conveniente usar os índices em forma de hash:

```
IDL> help,rh
RH              HASH  <ID=83  NELEMENTS=14>
IDL> print,rh
2455566: !NULL
2455564: !NULL
2455572: !NULL
2455575: !NULL
2455565: !NULL
2455574:            8
2455563:            0
2455567: !NULL
2455570:            3           4           5           6           7
2455576:            9
2455569:            1           2
2455573: !NULL
2455568: !NULL
2455571: !NULL
```

Cada elemento do hash **`rh`** é um (possivelmente vazio) array, que tem os índices dos elementos (no caso, do array **`obs.date`**) que caem no bin correspondente. Ex: na data **2455569** houve duas observações, as de índices **1** e **2** no array **`obs`**:

```
IDL> print,rh[2455569]
           1           2
IDL> print,obs[rh[2455569]]
{ HD59382     2455569 sl_0102.fits<ObjHeapVar8(HASH)>}
{ HD63281     2455569 sl_0110.fits<ObjHeapVar12(HASH)>}
```

Vetorização avançada (mas essencial) - inversão de índices

O mesmo tipo de inversão pode ser feito por arrays que não sejam inteiros:

No caso de faixas uniformes de valores reais, pode-usar diretamente um histograma, escolhendo-se os tamanhos dos bins.

No caso de reais únicos (buscando-se os valores *exatamente* iguais), faixas de valores não uniformes, strings, ou valores muito esparsos (ex: [1,1000,3007,3008,30010]), é necessário e/ou conveniente se trabalhar com índices inteiros para valores únicos.

No exemplo anterior, para evitar todos os bins vazios de várias datas seguidas sem observações:

```
IDL> obs=obs[sort(obs.date)]
IDL> u=uniq(obs.date)
IDL> print,u
           0           2           7           8           9
```

Normalmente, funções que identificam valores únicos esperam arrays ordenados

Índices dos elementos únicos

Datas únicas:

```
IDL> udates=obs[u].date
IDL> print,udates
     2455563     2455569     2455570     2455574     2455576
```

Usa-se então, no lugar do array original (`obs.date`), um array de índices para o original:

```
IDL> indexes=value_locate(udates,obs.date)
IDL> print,indexes
           0           1           1           2           2           2
    2           2           3           4
```

Vetorização avançada (mas essencial) - inversão de índices

Há um nível de indireção a mais, mas não há mais bins vazios (e é o que torna possível bins de tamanhos desiguais, bins reais ou de strings):

```
IDL>
h=histogram_pp(indexes,min=min(indexes),binsize=1L,reverse_list=rl,rev
erse_hash=rh,locations=locations)
IDL> print,rh
0:            0
1:            1           2
3:            8
2:            3           4           5           6           7
```

Convertendo os índices para índices do array original:

```
IDL> oindexes=list()
IDL> foreach el,rl do oindexes.add,indexes[el]
IDL> olocations=udates[locations]
IDL> foreach el,rl,i do print,olocations[i],' : ',obs[el].object
     2455563 : HD28985
     2455569 : HD59382 HD63281
     2455570 : HD63281 HD48561 HD78325 HR892561 HR748267
     2455574 : HR189365
     2455576 : HR167382
```

Escolha de contêiners

Qual o melhor para cada situação? Assim como para linguagens, não há “a melhor escolha”.

Cada pedaço do código vai se adequar mais a algumas escolhas, e menos a outras.

Muitas vezes, parte do processamento é melhor feita com um contêiner, e outra parte com outro: pode-se começar com um, e converter para outro no ponto em que o outro é melhor. (exemplos adiante)

Escolha de contêiners – pontos a considerar

Contêneirs dinâmicos e não homogêneos (em particular, listas e mapas) em geral armazenam referências aos elementos (como ponteiros).

Cópias destes contêiners por default são ***cópias rasas*** *(**shallow copies**)* - cópias das referências. Ex. (IDL):

```
IDL> l=list(1,2)
IDL> l2=l
IDL> print,l
       1
       2
IDL> print,l2
       1
       2
IDL> l2[0]=-1
IDL> print,l
      -1
       2
IDL> print,l2
      -1
       2
```

Para criar novas variáveis para os elementos, é necessário usar os métodos de ***cópias profundas*** *(**deep copies**)* – estes recursivamente entram em todos os elementos e fazem cópias profundas do que encontram.

Cada implementação costuma ter sua função / método que faz *deep copies*.

Escolha de contêiners – pontos a considerar

Contêiners implementados por objetos permitem facilmente gerar outros, por herança, com um comportamento ligeiramente diferente, necessário para alguma aplicação:

- Listas / mapas homogêneos (ou mapas de chaves homogêneas).
- Mapas que desconsideram maiúsculas/minúsculas nas chaves.
- Mapas que preservam a ordem de inserção dos elementos.
- Contêiners onde os dados ficam armazenados em disco, ou são obtidos por conexão de rede, ou são calculados sob demanda, mas que **apresentam a mesma interface que os contêiners usuais**: seu uso é indistinguível do uso dos contêiners “normais”.
- Mapas que armazenam listas: a cada atribuição, o elemento é adicionado à lista daquela chave, no lugar de substituir o que está presente lá:

Escolha de contêiners – pontos a considerar

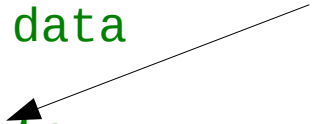
Ex. (IDL):

No lugar de

```
h=hash()
foreach element,files do begin
  read_spectrum,element,data

  if h.haskey(data.object_name) then $
   (h[data.object_name]).add,data else $
   h[data.object_name]=list(data)

endforeach
```

Se usa uma classe que mantém os elementos como listas, simplificando o uso:

```
h=hash_of_lists()
foreach element,files do begin
  read_spectrum,element,data

  h[data.object_name]=data

endforeach
```



Nesta hash, se já há um elemento daquela chave, ele é adicionado à lista daquela chave; se não, uma lista de um elemento é colocada naquela chave. (como acima)

Escolha de contêiners – pontos a considerar

Contêiners implementados por objetos permitem facilmente gerar outros, por herança, com um comportamento ligeiramente diferente, necessário para alguma aplicação:

Ex. (IDL): Mapas onde, na leitura, chaves são encontradas de forma não exata.

Por strings:

```
IDL> h=hash_by_string_match()
IDL> h['alpha centauri']=98
IDL> h['beta centauri']=200
IDL> h['gamma centauri']=100
IDL> h['alpha crux']=30
IDL> print,h['*centauri']
beta centauri:      200
gamma centauri:      100
alpha centauri:       98
```

Por reais:

```
IDL> spectrum=hash_by_nearest_real(read_spectrum('some_file.nc'))
IDL> print,spectrum
       6784.0000:       1.7852000
       6784.3000:       1.7852000
       6784.6000:       1.8134000
    (...)
IDL> print,spectrum[6784.1]
       1.7852000
IDL> print,spectrum[6784.2]
       1.7852000
```

Escolha de contêiners – pontos a considerar

Considerações de eficiência são muito dependentes de linguagem, das dimensões do problema, **e de como o código foi escrito**.

Com pequenas dimensões, para coisas que não são repetidas um número muito grande de vezes, estas diferenças são irrelevantes.

Mais relevante acaba sendo o que leva à melhor organização do código, e o que é mais confortável para o autor. E pode sair mais rápido usar algo que demora mais para executar, porque se levou um dia para escrever e testar o código, ao invés de uma semana.

Tipos primitivos (que não são objetos), como ponteiros, estruturas, arrays (em algumas linguagens), por não terem o *overhead* de chamadas de funções em objetos, nem todos os testes e infraestrutura dos contêiners de mais alto nível, tendem a ser mais eficientes para as operações que eles suportam.

Se para a variável em questão não são necessárias as operações de mais alto nível, pode ser então melhor escolher a opção de baixo nível, se a diferença for relevante.

Mesmo que se meça que uma forma é muito mais eficiente, há que se considerar se a diferença é relevante: uma forma que execute em 1 s é muito menos eficiente que uma que leve 1 ms. Mas se a operação só vai ser executada uma vez, qual é a diferença entre esperar 1s a mais para o programa terminar?

Já se o programa vai ser executado muitos milhares de vezes, uma diferença de 1s pode ficar relevante (um dia tem menos de 10^5 s).

Escolha de contêiners - listas x arrays

Listas e arrays são os principais contêiners para acesso por índices, e por isso compartilham muitos usos.

Principais pontos para decidir entre eles; **os mesmos que são vantagens em alguns usos são desvantagens em outros:**

- Listas são dinâmicas, 1D e podem ser heterogêneas
- Arrays são estáticos, homogêneos e podem ser mais de 1D

Em geral,

- Listas são mais convenientes quando é necessário:
  - “arrays não regulares”
  - adicionar / remover elementos (incluindo quando não se sabe antecipadamente quais serão os elementos)
  - elementos não escalares ou que não sejam do mesmo tipo
  - modificar o comportamento do contêiner* (por herança de classes).
- Arrays são mais convenientes quando é necessário:
  - mais de 1D (quando disponível)
  - operações vetoriais (quando disponível)
  - elementos escalares e homogêneos.

*Em algumas linguagens (ex: C++, Java, Python+Numpy), os arrays mais comuns podem já ser classes.

Escolha de contêiners - listas x arrays

Implementações variam em ter checagem de índices para arrays: as de alto nível os testam, ao contrário dos arrays primitivos de C, C++, Fortran.

Se a situação não exige as comodidades de listas, arrays costumam ser mais vantajosas, pela semântica vetorial, além de serem mais eficientes.

Escolha de contêiners - listas x arrays

É comum que a escolha mais cômoda seja usar uma lista durante a sua formação (elementos sendo adicionados / removidos), e converter o resultado para array, para depois fazer uso da semântica vetorial.

Se tem o melhor dos dois mundos, quando cada tipo de facilidade é necessária.

Modificando o exemplo anterior, onde observações eram colocadas em uma lista. Ex. (IDL):

```
files=file_search('./*.fits')
objects=list()
for i=0,n_elements(files)-1 do begin
  read_spectrum,files[i],wavelength,flux,instrument,name
  objects.add,{name:name,wavelength:wavelength,flux:flux,$
    instrument_data:instrument_data,...})
endfor
(..)
i=0
foreach element,objects,i do $
  if do_not_keep(element.wavelength,element.flux) then
objects.remove,i
```

Escolha de contêiners - listas x arrays

Então, depois de removidos os espectros indesejáveis, pode-se continuar com um array:

```
IDL> kept_objects=objects.toarray()
IDL> help,kept_objects
KEPT_OBJECTS    STRUCT    = -> <Anonymous> Array[4]

IDL> help,kept_objects[0]
** Structure <7410a3d8>, 4 tags, length=16440, data length=16440, refs=3:
   NAME            STRING    'HD3905'
   WAVELENGTH      DOUBLE    Array[1024]
   FLUX            DOUBLE    Array[1024]
   INSTRUMENT_DATA STRUCT    -> <Anonymous> Array[1]

IDL> help,kept_objects[0].instrument_data
** Structure <741d8278>, 3 tags, length=40, data length=40, refs=2:
   EXPOSURE_TIME   DOUBLE           1000.0000
   DATE            STRING    '20090130'
   INSTRUMENT_NAME STRING    'SE_5'

IDL> print,kept_objects.name
HD3905 HD5298 HD4859 HR36205

IDL> print,kept_objects.instrument_data.date
20090130 20090208 20090208 20090209

IDL> help,kept_objects.instrument_data.exposure_time
<Expression>    DOUBLE    = Array[4]

IDL> print,kept_objects.instrument_data.exposure_time
       1000.0000       1200.0000       2000.0000       1000.0000
```

Escolha de contêiners - estruturas x mapas

Estruturas e mapas são os principais contêiners para acesso por chaves (nomes), e por isso compartilham muitos usos.

Principais diferença:

- Mapas são dinâmicos
- Estruturas são estáticas

Em algumas implementações também é possível acessar os elementos por índices (ex: estruturas em IDL, *named lists* em R).

Em geral,

- Mapas são mais convenientes:
  - quando não se sabe antecipadamente quais serão todos os campos
  - quando os campos podem precisar mudar de dimensões / tipo
  - quando será necessário adicionar / remover campos
  - quando se precisa de chaves que não são strings, ou não são strings simples
  - quando se quer modificar o comportamento do contêiner (por herança de classes).

- Estruturas são mais convenientes:
  - quando se quer combinar os resultados em arrays (arrays de estruturas ou estruturas de arrays), o que em geral permite melhor semântica vetorial para acessar os campos
  - quando não se vai mais alterar o número / tipo / dimensões de elementos.

Se a situação não exige as comodidades de mapas, estruturas podem ser mais vantajosas, pela semântica vetorial, além de serem mais eficientes.

Escolha de contêiners - estruturas x mapas

É comum que a escolha mais cômoda seja usar um mapa (possivelmente mapa de listas ou lista de mapas) durante a sua formação (elementos sendo adicionados / removidos), e converter o resultado para estrutura (possivelmente array de estruturas ou estrutura de arrays) para depois fazer uso da semântica vetorial.

Se tem o melhor dos dois mundos, quando cada tipo de função é necessária.

- Comum, por exemplo, na leitura de um arquivo com várias variáveis, de tipos e dimensões diferentes (parâmetros para um modelo, resultados de um modelo, cabeçalho de arquivo de dados (fits, cub)).
- Ao começar a ler o arquivo, não se sabe quantos campos vão ser encontrados, nem seus nomes, tipos e dimensões (necessários para definir uma estrutura no começo).

Exemplo: armazenar todos os dados de uma observação, incluídos no cabeçalho fits:

```
SIMPLE  =                    T /image conforms to FITS standard
BITPIX  =                   32 /bits per data value
NAXIS   =                    2 /number of axes
NAXIS1  =                 1024 /
NAXIS2  =                 1024 /
BSCALE  =              1.00000 / Scaling factor
BZERO   =              0.00000 / Scaling zero-point
TELESCOP= 'Keck II           ' / Telescope
SLITNAME= '0.041x2.26        ' / Slit Name
SLITPA  =              351.979 / Slit position angle
SLITX   =              131.500 / X pixel of the center of the slit
SLITY   =              125.000 / Y pixel of the center of the slit
SLITANG =               13.900 / Slit angle in the scam. CCW from hor.
LST     = '12:01:04.46       ' / local apparent sidereal time
(...)
```

Escolha de contêiners - estruturas x mapas

Com mapas, é fácil processar um a um os elementos do arquivo, os adicionando conforme forem encontrados.

- Algo especialmente relevante quando não se sabe previamente os tipos e as dimensões das variáveis (no caso de cabeçalhos fits, todas são escalares; mas em outros formatos é comum que haja arrays também).

Para o exemplo fits anterior, poderia ser (IDL, com o uso da biblioteca IDLAstro):

```
file_data=readfits(file,file_header)
header_values=hash()
header_descriptions=hash()
foreach element,iheader do begin
  parse_header_line,element,key,value,description
  header_values[key]=value
  header_descriptions[key]=description
endforeach
```

parse_header_line acima é uma rotina que toma uma linha, como string, do tipo

```
“NAXIS   =                    2 /number of axes”
```

E retorna em três variáveis separadas o nome (**NAXIS**), o valor (**2**) e o comentário (**number of axes**).
O valor é retornado como string, inteiro ou double. Esta rotina será mostrada na aula de processamento de strings.

Escolha de contêiners - estruturas x mapas

O resultado, como hashes é

```
IDL> help,header_values,header_descriptions
HEADER_VALUES   HASH  <ID=1  NELEMENTS=194>
HEADER_DESCRIPTIONS HASH  <ID=3  NELEMENTS=194>

IDL> print,header_values
SLITANG:        13.900000
MJD-OBS:        54087.691
FILENAME: 'dec18s0041.fits   '
(...)

IDL> print,header_descriptions
SLITANG:  Slit angle in the scam. CCW from hor.
MJD-OBS:  modified julian date of observ
FILENAME:  Name of file
(...)
```

Que podem ser convertidos para estruturas, pela sua conveniência para uso vetorial:

```
IDL> s_header_values=header_values.tostruct()
IDL> help,s_header_values
** Structure <7027cae8>, 194 tags, length=2360, data length=2360, refs=1:
   ROTMODE         STRING    ''position angle    ''
   BSCALE          DOUBLE           1.0000000
   LSAMPPWR        STRING    ''0.000000          ''
   XENON           LONG64                        1
(...)
```

Outras formas para criar a estrutura (no caso, de 194 campos), sem previamente saber o número, tipo e dimensões dos campos, seriam trabalhosas e ineficientes.

Sumário

4 – Slides em http://www.ppenteado.net/pea/pea03_conteiners.pdf

- Contêiners
- Arrays
- Listas
- Mapas
- Outros contêiners
- Vetorização
- Escolha de contêiners

Exercícios sugeridos:

- Vetorizar a solução do exercício anterior:
  Escrever um programa para calcular a integral de uma gaussiana:

$$\int_{-\infty}^{\infty} e^{-x^2} dx = \sqrt{\pi}$$

  Da forma mais simples: por áreas de retângulos de largura constante.

  É possível calcular esta integral sem loops? Em menos de uma dúzia de linhas de código?

- Acompanhar os dois últimos slides desta aula sobre inversão de índices (para casos não contíguos ou não-inteiros).

Mais exemplos de usos dos contêiners em leitura de arquivos (próxima aula).

Próxima aula

5 – Slides em http://www.ppenteado.net/pea/pea04_strings_io.pdf

- Strings
- Expressões regulares
- Arquivos

http://www.ppenteado.net/pea