Programación para Física y Astronomía

Departamento de Física.

Profesores: Claudia Loyola/ Joaquín Peralta
Felipe Moreno/ Fabián Gómez-Villafañe

Primer Semestre / 2023

Universidad Andrés Bello

Universidad Andrés Bello®

Resumen

Listas y Arreglos de datos

Listas

- En física es común para una variable representar varios números a las vez:
  - Un vector $\vec{r}$ que representa la posición de un punto en el espacio tridimensional.
  - Una matriz, denotada por ejemplo por M, puede representar una cuadrícula de $m \times n$ números.
- En Python existe una característica estándar para guardar una colección de números llamada **contenedores** (containers). Existen varios tipos de contenedores, en esta clase veremos las **Listas** y **Arreglos**, y algo más.
- **Listas**: Son una lista de cantidades, una después de otra, de cualquier tipo conocida para Python (hasta ahora para ustedes, enteros, floats, complejos y arreglos).
- Las cantidades en una lista son llamadas *elementos* y pueden o no ser del mismo tipo.

Listas

- Una lista en Python se escribe:

```
[3, 0, 0, -7, 24]
```

- Los elementos son separados por comas.
- Todos los elementos de una lista deben estar contenidos en corchetes cuadrados "*[ ]*".

```
#Una lista puede ser asignada a una variable:
r=[1,1,2.5,3+4.6j]
```

- Las cantidades que constituyen los elementos de una lista pueden ser especificados usando otras variables.

```
x, y, z=1.0, 1.5, -2.2
r=[x,y,z]
```

A tener en cuenta

Python siempre evalúa la expresión antes de la asignación:

```
1  x=1.0
2  y=1.5
3  z=-2.2
4  r=[x,y,z]
5  print(r)
6  x=5.0
7  print(r)
```

salida 1: Incorrecto

```
1  [1.0,1.5,-2.2]
2  [5.0,1.5,-2.2]
```

salida 2: Correcto

```
1  [1.0,1.5,-2.2]
2  [1.0,1.5,-2.2]
```

Elementos de una Lista

- Los elementos individuales de una lista r pueden ser accesados mediante: *r[0], r[1], r[2],...,r[n-esimo]*.
- Los elementos están enumerados en orden creciente desde el principio hasta el final de la lista comenzando en **cero**. Los elementos individuales de una lista se invocan usando el nombre de la variable seguida de "*[]*" entre los que se indica el número o índice del elemento.
- Los elementos individuales, como *r[0]* pueden ser usados de la misma forma que una variable:

```
1  from math import sqrt
2  r=[1.0, 1.5, -2.2]
3  modulo=sqrt(r[0]**2+r[1]**2+r[2]**2)
4  print(modulo)
```

Elementos de una Lista

- Podemos modificar valores de elementos individuales en cualquier posición de la lista y cuando sea necesario:

```
1 r=[1.0, 1.5, -2.2]
2 r[1]=3.5
3 print(r)
```

```
1 [1.0, 3.5, -2.2]
```

- Funciones integradas: el poder de Python

```
1 r=[1.0, 1.5, -2.2]
2 total= sum(r)
3 print(total)
4 promedio=sum(r)/len(r)
5 print(promedio)
```

```
1 0.3
2 0.1
```

Funciones Integradas

- La función *map(function,iterable)* permite aplicar una función ordinaria a todos los elementos de una lista y almacenar el resultado en memoria para ser usado como se desee.

```
1 from math import log
2 r=[1.0, 1.5, 2.2]
3 logr=list(map(log,r))
```

Funciones Integradas

- Las listas en Python permiten la posibilidad de agregar elementos a una lista existente, por ejemplo a la lista *r*, usando:

```
1  r.append(6.1)
```

- La expresión anterior consiste de el nombre de nuestra lista, *r*, seguido de un punto "**.**", y luego ***append(6.1)***, que corresponde a un Método del Objeto *lista*. Esta declaración permite agregar un nuevo elemento al final de la lista. El valor puede ser una variable o expresión matemática.

```
1  r=[1.0, 1.5, 2.2]
2  x=0.8
3  print(r)
4  r.append(2*x+1)
5  print(r)
```

```
1  [1.0, 1.5, 2.2]
2  [1.0, 1.5, 2.2, 2.6]
```

Lista *empty*

Nosotros podemos crear listas vacías, es decir sin elementos, con la finalidad de agregar elementos uno por uno y hacer crecer nuestra lista tanto como queramos:

```
1 r=[]
2 r.append(1.0)
3 r.append(1.5)
4 r.append(-2.2)
```

También podemos remover valores desde el final de una lista usando el método *pop( )* de forma similar a *append( )*:

```
1 r=[1.0, 1.5, -2.2, 2.6]
2 r.pop()
```

Lista empty

Además, podemos remover un elemento cualquiera de la lista usando *pop(i)* donde el argumento *i* se refiere al i-ésimo elemento de la lista:

```
1  #Resumen append y pop
2  r=[]
3  r.append(1.0)
4  r.append(1.5)
5  r.append(-2.2)
6  r.append(2.6)
7  print(r)
8  r.pop()
9  print(r)
10 r.pop(1)
11 print(r)
```

```
1  #salida
2  [1.0, 1.5, -2.2, 2.6]
3  [1.0, 1.5, -2.2]
4  [1.0, -2.2]
```

Más operaciones sobre Listas

Método	Descripción
`L=[0, 1, 2, 3]`	Se define una lista. `L`
`L.extend([-2.2,8,9])`	`extend(iterable)` adiciona al final de la lista los elementos pasados como argumentos.
`L.insert(1,-2.2)`	`insert(i,value)` inserta un elemento en la posición i.
`L.index(-2.2)`	`index(value)` busca el valor pasado como argumento y retorna su posición (la primera ocurrencia).
`L.count(-2.2)`	`count(value)` retorna el número de ocurrencias del valor pasado como argumento.
`L.sort()`	`sort()` ordena los elementos de la lista
`L.reverse()`	`reverse()` revierte los elementos de la lista
`L.remove()`	`remove(value)` elimina el primer elemento que concuerde con el valor pasado como argumento.

Arreglos

Arreglos

- Muy similar a las Listas, pero con dos diferencias importantes:
  1. El número de elementos en un arreglo es **fijo**. No se pueden agregar o quitar elementos una vez que ha sido creado.
  2. Los elementos de un arreglo deben ser todos del mismo tipo. No se puede cambiar el tipo de elemento una vez creado.
- ¿Por qué **Arreglos**, si al parecer la Lista es más versátil?
  1. Los arreglos pueden ser n-dimensionales. Listas son solo unidimensionales.
  2. Arreglos se comportan más o menos como vectores o matrices: se pueden realizar operaciones aritméticas sobre ellos.
  3. Arreglos trabajan más rápido que las listas.
- Los arreglos tienen mucha aplicabilidad en Física: en Física a menudo se trabaja con un número fijo de elementos de un mismo tipo.

Arreglos

- Para crear arreglos es necesario indicarle al computador cuántos elementos y de qué tipo es lo que vamos a definir. Las funciones que permiten realizar estas operaciones se encuentran en el paquete *numpy*.
  Ejemplo: para crear un arreglo unidimensional de tamaño *n*, elementos todos inicialmente iguales a cero, podemos usar la función zeros del paquete *numpy*.

```
1 from numpy import zeros
2 a=zeros(4,float)
3 print(a)
```

Notar que la salida no posee comas entre los elementos, solo espacios en blanco.

Arreglos

```
1 #Otros arreglos
2 from numpy import zeros
3 a=zeros(10,int)
4 a=zeros(100,complex)
```

El tamaño de los arreglos que se pueden crear es limitado solo por el tamaño de la memoria en el computador (máquinas modernas pueden crear arreglos de incluso billones de elementos).

Arreglos

- Para crear arreglos bidimensionales con *m* filas y *n* columnas, usando la función *zero()*, se escribe: *zeros([m,n],float)*.

```
1 a=zeros([3,4],float)
2 print(a)
```

```
1 [[0. 0. 0. 0.]
2  [0. 0. 0. 0.]
3  [0. 0. 0. 0.]]
```

Función	Descripción
ones([3,4],float)	*ones()* crea un arreglo con elementos iguales a 1.
empty([3,4],float)	*empty()* crea un arreglo vacío.
r=[1.0,1.5,-2.2] *a=array(r,float)*	Declaración. *array()* crea un arreglo a partir de una lista.

Arreglos

Para crear arreglos bidimensionales con *m* filas y *n* columnas.

```
1 a=array([[1,2,3],[4,5,6]],int)
2 print(a)
```

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

Referenciando elementos individuales del arreglo:

- Arreglo unidimensional: *a[0]*, *a[1]*, *a[2]=4*.
- Arreglo bidimensional: *a[2,4]*.

```
1 from numpy import zeros
2 a=zeros([2,2],int)
3 a[0,1]=1
4 a[1,0]=-1
5 print(a)
```

```
1 [[ 0 1]
2  [-1 0]]
```

Nota

Los arreglos bidimensionales en Python son impresos usando la convención de aritmética matricial estándar: primer índice denota filas y segundo columnas.

Lectura de archivos

- La función *loadtxt()* del paquete *numpy* permite leer un conjunto de valores desde un archivo en el computador:

valores1.txt

```
1.0
1.5
-2.2
2.6
```

```
1 from numpy import loadtxt
2 a=loadtxt("valores1.txt",float)
3 print(a)
```

```
1 [1.0 1.5 -2.2 2.6]
```

Nota: el archivo debe estar en el mismo directorio donde se ejecuta el programa. En caso contrario se debe especificar la ruta al fichero.

- Usando el mismo programa se puede leer otro archivo:

valores2.txt

```
1 2 3 4
3 4 5 6
6 7 8 9
```

```
1 [[ 1. 2. 3. 4.]
2  [ 3. 4. 5. 6.]
3  [ 6. 7. 8. 9.]]
```

Aritmética con arreglos

Los elementos individuales se comportan como variables ordinarias, por lo tanto podemos hacer operaciones naturalmente:

```
a[0]=a[1]+1
x=a[2]**2-2*a[3]/y
```

```
#aritmética sobre todo el arreglo
from numpy import array
a=array([1,2,3,4],int)
b=2*a
print(b)
```

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

Nota: Este tipo de operaciones requiere arreglos del mismo tamaño. En caso contrario se produce un error.

Aritmética con arreglos

Multiplicación de arreglos:

```
1 a=array([1,2,3,4],int)
2 b=array([2,4,6,8],int)
3 print(a*b)
```

La multiplicación "*" en Python se realiza elemento por elemento. **La multiplicación no es el producto punto**. La división "/" funciona de forma similar. La función *dot( )*, del paquete *numpy* permite efectuar el producto punto sobre arreglos:

```
1 from numpy import array,dot
2 a=array([1,2,3,4],int)
3 b=array([2,4,6,8],int)
4 print(dot(a,b))
```

Aritmética con arreglos

- Todas las operaciones anteriores trabajan con arreglos bidimensionales:

```
1 a=array([[1,3],[2,4]],int)
2 b=array([[4,-2],[-3,1]],int)
3 c=array([[1,2],[2,1]],int)
4 print (dot(a,b)+2*c)
```

```
1 [[-3, 5]
2  [ 0, 2]]
```

Corrobore los resultados.

- También se puede operar usando matrices y vectores: supongamos que *v* es un arreglo unidimensional y *a* es una matriz.
  1. *dot(a,v)*: trata el vector *v* como vector columna. Multiplica *v* por la izquierda con *a*.
  2. *dot(v,a)*: trata el vector *v* como vector fila. Multiplica *v* por la derecha con *a*.
  3. Las funciones sum, max, min, len y map pueden ser aplicadas en arreglos n-dimensionales.

Aritmética con arreglos

- Funciones *size* y *shape*, muy útiles para arreglos de n-dimensiones.
  1. *size*: número total de elementos en todas las filas y columnas del arreglo.
  2. *shape*: lista las dimensiones de un arreglo en cada eje (ejemplo de tupla).

```
1 a=array([[1,2,3],[4,5,6]],int)
2 print(a.size)->6
3 print(a.shape)->(2, 3)
```

A tener en cuenta

Considere el siguiente programa:

```
1 from numpy import array
2 a=array([1,1],int)
3 b=a
4 a[0]=2
5 print(a)
6 print(b)
```

```
1 [2, 1]
2 [2, 1]
```

Esta salida es **inesperada** considerando lo aprendido en la sección de Listas.

En Python la asignación directa de arreglos de esta forma no funciona, debido a que Python solo crea una nueva variable “b” donde se copia el nombre del arreglo “a”. Digamos que los nombres de los arreglos almacenan la posición en memoria del computador donde se almacenó la información. De este modo a y b apuntan al mismo arreglo en memoria del computador.

A tener en cuenta

La función *copy()* puede realizar esta tarea si fuese necesario:

```
1 from numpy import array, copy
2 a=array([1,1],int)
3 b=copy(a)
4 a[0]=2
5 print(a)
6 print(b)
```

```
1 [2, 1]
2 [1, 1]
```

Recorte de datos

El recorte (slicing) es otro recuso muy útil en Python. Funciona en arreglos y listas y permite obtener fragmentos o secciones de un arreglo o lista.

```
1 r=[1,3,5,7,9,11,13,15]
2 s=r[2:5]
3 print(s)
```

```
1 [5,7,9]
```

`r[m:n]` es otra lista compuesta de un subconjunto de elementos de `r`, comenzando con el elemento en la posición `m` y llegando hasta la posición `n` pero sin incluir dicho elemento n-ésimo.

Variantes

- `r[2:]` : todos los elementos de la lista desde el elemento 2 hasta el final.
- `r[:5]` : todos los elementos de la lista desde el comienzo hasta el elemento 5 sin incluirle.
- `r[:]` : todos los elementos de la lista desde el comienzo al final.

Recorte de datos

El recorte en **Arreglos** funciona del mismo modo que con un Lista:

```
1 #array slicing
2 from numpy import array
3 a = array([2,4,6,8,10,12,14,16],int)
4 b = a[3:6]
5 print(b)
```

```
1 [8 10 12]
```

Recorte de datos

El recorte en **Arreglos** bidimensionales:

array a

```
[[ 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]]
```

```
1 print(a[2,3:6])
```

```
1 [16 17 18]
```

a[2,3:6] retorna un arreglo unidimensional con tres elementos iguales a *a[2,3]*, *a[2,4]* y *a[2,5]*.

Recorte de datos

Relacionado con el ejemplo anterior y la definición del arreglo *a*:

- `a[2:4,3:6]` : retorna un arreglo bidimensional de tamaño 2×3.

  ```
  [[16 17 18]
  [22 23 24]]
  ```

- `a[2,:]` : retorna la fila 2 completa del arreglo a.
  ```
  [13 14 15 16 17 18]
  ```

- `a[:,3]` : retorna la columna 3 completa del arreglo a.
  ```
  [ 4 10 16 22 28]
  ```

Bucle o ciclo “FOR”

Bucle o ciclo “FOR”

Dejamos la estructura de control “FOR” para esta sección debido a que presenta inmejorables ventajas para el uso con **contenedores** como **Listas** y **Arreglos**.

- El bucle “FOR”, al igual que “WHILE”, permite recorrer los elementos de una lista o arreglo uno por uno.

```
1 r=[1, 3, 5]
2 for n in r:
3     print(n)
4     print(2*n)
5 print("Fin")
```

```
1 1
2 2
3 3
4 6
5 5
6 10
7 Fin
```

Bucle o ciclo “FOR”

El parámetro **end** de la función *print()* especifica qué se imprimirá al final de la salida. Por defecto, es un salto de línea (`'\n'`), por lo que la siguiente impresión será en una línea nueva. Sin embargo, podemos modificar el valor del parámetro end para cambiar este comportamiento. Por ejemplo, si queremos imprimir la salida en la misma línea, podemos establecer que **end** es igual a un espacio en blanco (`' '`) o simplemente una cadena vacía (`''`).

- En un ciclo “FOR”, podemos utilizar el parámetro **end** para controlar cómo se imprime cada elemento. Por ejemplo, si queremos imprimir todos los elementos en una misma línea separados por un espacio, podemos hacerlo de la siguiente manera:

```
1 for i in range(5):
2     print(i, end=' ')
```

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

Funciones útiles para el ciclo “FOR”

A continuación se muestra una tabla con las funciones más usadas en el ciclo “FOR”. Estas funciones se encuentran disponibles en el paquete numpy de Python.

Función	Descripción
`range(n)`	crea una lista de enteros comenzando en cero y de largo n.
`range(i,j,k)`	crea una lista de enteros comenzando en `i`, terminando en `j` y con un paso de tamaño `k`.
`arange(n)`	funciona muy similar a `range()` pero genera **arreglos**. Puede ser usado con uno, dos o tres argumentos también.
`linspace(i,j,k)`	(`numpy`) genera arreglos con `k` valores tipo `float` entre `i` y `j`. Note que `linspace()` incluye el último punto en el rango.

Ordenando Datos

Ordenando Datos

Python posee un Método integrado especial para realizar ordenamiento de datos en Listas, llamado *sort()*. Sin embargo, este método no nos permite ordenar según algún criterio parcial o arbitrario una lista cualquiera. Por ejemplo, ordenar la siguiente lista según el promedio de las listas anidadas.

```
1  L=[[1, 200, 50000],
2  [1, 2, 3],
3  [1],
4  [100],
5  [2, 2, 3, 4, 5, 6, 3, 21],
6  [1, 2, 5, 2, 5, 7],
7  [1, 2, 0, 8, 7, 6],
8  [5, 6, 4, 8, 3, 2],
9  [2, 3],
10 [1, 2, 99]]
```

Actividad a realizar hoy

Ordenando Datos

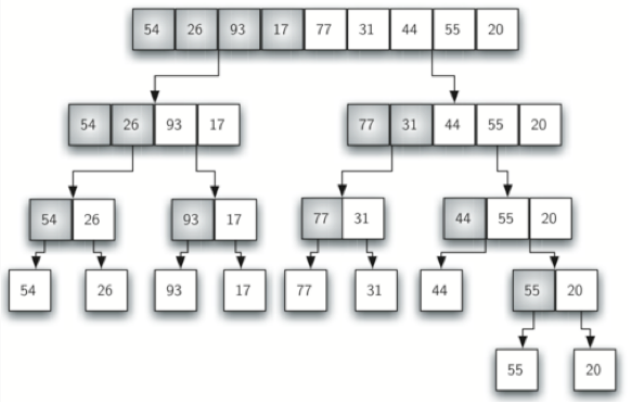
Para resolver este tipo de problemas es necesario conocer los algoritmos de orden. Existen varios entre los cuales destacan: **Insertion**, **Merge**, **Heapsort** and **Quicksort**, debido a que son los más eficientes y usados (*sort()* de Python usa Timsort, un mix de Insertion and Merge).

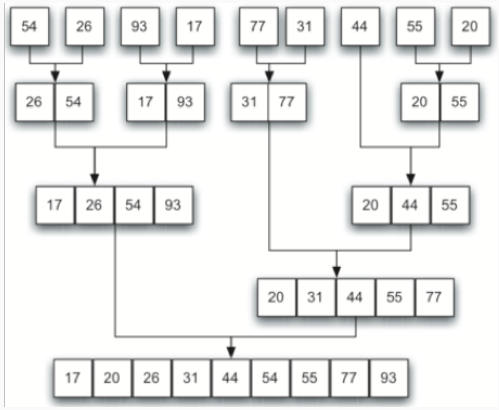
- Insertion: Es quizás uno de los algoritmos más simples e intuitivos para el ser humano. El orden final se obtiene por trabajar con un ítem a la vez:

```
1  def insertionSort(lista):
2    for i in range(1,len(lista)):
3      valoractual=lista[i]
4      posicion=i
5      while posicion>0 and lista[posicion-1]>valoractual:
6        lista[posicion]=lista[posicion-1]
7        posicion=posicion-1
8      lista[posicion]=valoractual^^I^^I
9  milista=[54,26,93,17,77,31,44,55,20]
10 insertionSort(milista)
```

Ordenando Datos

- Merge sort: algoritmo basado en la estrategia de divide y vencerás. Algoritmo recursivo que continuamente divide una lista en mitades:
  - Si la lista está vacía o tiene un ítem, está ordenada por definición.
  - Si la lista posee más de un ítem, se divide la lista y recursivamente se invoca al algoritmo nuevamente.
  - Una vez que las dos mitades están ordenadas, se desarrolla la operación de merge(mezcla), mezclando las listas y retornando la lista ordenada.





Ordenando Datos

```
1  def mergeSort(lista):
2    print("Dividiendo ",lista)
3    if len(lista)>1:
4    centro = len(lista)//2
5    mit_izq = lista[:centro]
6    mit_der = lista[centro:]
7    mergeSort(mitad_izq)
8    mergeSort(mitad_der)
9    i=0
10   j=0
11   k=0
12   while i<len(mit_izq) and j<len(mit_der):
13     if mitad_izq[i] < mitad_der[j]:
14       lista[k]=mitad_izq[i]
15       i=i+1
16     else:
17       lista[k]=mitad_der[j]
18       j=j+1
19     k=k+1
```

```
20   while i < len(mitad_izq):
21     lista[k]=mitad_izq[i]
22     i=i+1
23     k=k+1
24   while j < len(mitad_der):
25     lista[k]=mitad_der[j]
26     j=j+1
27     k=k+1
28   print("Mezclando ",lista)
29 milista = [54,26,93,17,77,31,44,55,20]
30 mergeSort(milista)
31 print(milista)
```

Tuplas y Diccionarios

Tuplas y Diccionarios

Las **Tuplas** son muy parecidas a las Listas con la excepción de que son inmutables (no puede modificarse) una vez creadas.

- Se definen como una lista, salvo que los elementos se encierran entre paréntesis “()”.
- Los elementos de una tupla tienen orden definido y pueden ser de cualquier tipo.
- No se puede añadir (append) o eliminar (pop) elementos en una tupla.

Ejemplo	Definición
`T=(0,)`	definición de una tupla con 1 elemento.
`T=(0,'Ni',1.2,3)`	definición de una tupla con 4 elementos.
`T=('abs',('def','ghi'))`	tupla de 2 elementos, string y tupla.
`T[i]`	elemento i-ésimo de la tupla.
`T1 + T2`	concatenación de tuplas.

Tuplas y Diccionarios

- Los Diccionarios son muy similar a las listas (indexadas por un rango de valores), con la diferencia que los diccionarios son indexados por llaves (keys), las que deben ser de tipo inmutable (string, números, tuplas).
- Se dice que los diccionarios son un conjunto desordenado de pares llave:valor, con el requerimiento que las llaves sean únicas.
- Los diccionarios se definen usando una lista de pares llave:valor separados por comas y encerrado todo el conjunto por paréntesis-llave "*{ }*".
- La principales operaciones en un diccionario son: guardar un valor con una llave y extraer algún valor dada la llave.

Tuplas y Diccionarios

Ejemplo	Definición
`D={}`	diccionario vacío.
`D={'pan':2, `huevos':3}`	diccionario con dos items.
`D={'comida':{'pan':2, `huevos':3}}`	diccionarios anidados.
`D=dic(name='Pepe',age=25)`	definición usando la función `dic()`.
`D=dic.(zip(keyslist,valslist))`	definición usando tuplas.
`D['pan']->2`	valor asociado a la llave 'pan'.
`D['comida']['pan']->2`	valor asociado a 'comida' y 'pan' anidados.

Actvidades

- Implemente lo métodos *`InsertionSort`* y *`MergeSort`*, de las slides 32 y 34.
- Utilizando *`sort()`* de Python, ordene la lista según se indica en el slide 31.
- Genere 2 matrices de tamaño 3x3 con número enteros al azar entre 1 y 10. Luego multiplique estas matrices, compruebe el resultados usando lápiz y papel. Puede averiguar –en internet– como generar los números al azar.

Fin