Recuerdos fundamentales Numpy.

A lo largo de este curso se utilizaran diferentes algortimos para encontrar (por ejemplo) aproximaciones de soluciones de ecuaciones diferenciales o sistemas de ecuaciones no lineales. Estos algoritmos requieren el manejo básico arreglos y por tanto el uso de la librería NumPy es fundamental. En esta clase inicial, repasaremos las nociones fundamentales de dicha librería y utilizaremos algunos comandos básicos.

El objeto `ndarray` (N-dimensional array) es el objeto principal para representar arreglos de datos. Este objeto puede manejar arreglos multidimensionales de cualquier tamaño (restringido a la memoria del ordenador).

- Un `ndarray` tiene un tamaño fijo. Para modificar su tamaño hay que crear un nuevo arreglo.
- Un `ndarray` permite operaciones "elemento por elemento"

In [ ]:
```
import numpy as np #esto significa que abreviamos numpy por np
```

In [ ]:
```
# Creamos un arreglo
arr = np.array([1,2,3])
# Arreglos especiales
a = [1,2,3] #listas
b = [1,1,1]
c = np.array([1,1,1])
```

In [ ]:
```
arr
```

In [ ]:
```
a+c
```

In [ ]:
```
a + b
```

In [ ]:
```
arr + c
```

In [ ]:
```
a
```

In [ ]:
```
arr + a
```

Algunas de las propiedades de un arreglo se pueden obtener con las siguientes funciones:

In [ ]:
```
print(arr.shape) #entrega cantidad de dimensiones y de elementos en ellas,
print(arr.ndim) #numero de dimensiones de un arreglo
print(arr.size) #cantidad de entradas que tiene el arreglo
```

In [ ]:
```
q = np.array([[1,2],[3,4],[1,3]])
q
r = np.array([1,2,3])
```

In [ ]:
```
q
```

In [ ]:
```
q.size
```

In [ ]:
```
q.shape
```

In [ ]:
```
q.ndim
```

In [ ]:
```
r.ndim
```

Arreglos especiales:

In [ ]:
```
# Creamos un arreglo lleno de ceros.
t=np.zeros((4, 10))
t
```

In [ ]:
```
# Creamos un arreglo lleno de 1's
np.ones((2,10))
```

In [ ]:
```
# arreglo lleno de una constante
np.full((3,10), 3.14)
```

In [ ]:
```
# arreglo vacio
np.empty(11) #lo genera con basura
```

In [ ]:
```
#arreglo aleatorio de números enteros entre el 0 y 9
np.random.randint(0,10,100) #(valor inicial, valor final+1, tamaño)
```

In [ ]:
```
# arreglo con números desde el 5 al 30 con saltos de a 3
np.arange(5,30,3)
```

In [ ]:
```
#arreglo equiespaciado de 0 a 1 con 11 puntos
np.linspace(0,1,11)
```

Operaciones

Podemos usar las operaciones comunes. Ojo que se hacen término a término

In [ ]:
```
a = np.array([1,2,3])
b = np.array([1,-1,2])
print(a+b)
print(a-b)
print(a*b)
print(a/b)
```

In [ ]:
```
print(np.exp(a))
print(np.sqrt(a))
print(np.sin(b))
print(np.log(a))
```

Recorrer Arreglos

Para recorrer un arreglo podemos usar los siguientes programas que son equivalentes:

In [ ]:
```
arr=np.array([1,2,4,6,7,8,9,98,76,98, 65])
```

In [ ]:
```
arr[-2]
```

In [ ]:
```
arr[9]
```

In [ ]:
```
arr[0]
```

In [ ]:
```
np.size(arr)
```

In [ ]:
```
for i in np.arange(np.size(arr)):
    print(arr[-1-i])
```

In [ ]:
```
w = ['hola', 'mundo', 'chao']
len(w) #entrega número de items en una lista
```

In [ ]:
```
len(arr)
```

In [ ]:
```
for i in arr:
    print(i)
```

Algoritmos en arreglos

Vamos a implementar un par de algoritmos en arreglos.

Dado un arreglo `a` diremos que la posición `k` es un "peak" si `a[k-1]` $\leq$ `a[k]` y `a[k+1]` $\leq$ `a[k]` .

Problema: Dado un arreglo `a` encontrar un peak de `a`

Buscaremos un peak del arreglo a, considerando que los extremos del arreglo no tienen vecinos.

In [ ]:
```
#Primera idea: Recorrer el arreglo y verificar si es un peak
a = np.array([5,2,3,2,3,2,3,4])

def buscar_peak(a):
    for i in np.arange(1,len(a)-1):# recorrer el arreglo
        if a[i] >= a[i-1] and a[i] >= a[i+1]:#verificar si es un peak
            print('La posición', i, ' es un peak')

buscar_peak(a)
```

In [ ]:
```
b = np.random.randint(0,100,1000)
print(b)
buscar_peak(b)
```

Matrices en Numpy

In [ ]:
```
#Creamos una matriz de 2x3:
A = np.array([[1, 2, 3], [3, 4, 5]])
print(A)
```

In [ ]:
```
print(A.shape)
print(A.ndim)
print(A.size)
print(len(A))
```

Acceder a los elementos de la matriz:

In [ ]:
```
#Tercer elemento de la segunda fila:
print(A[1,2])
```

In [ ]:
```
#Último elemento de la primera fila
A[0,-1]
```

In [ ]:
```
A[1][2]
```

In [ ]:
```
#Primera fila:
A[0]
```

In [ ]:
```
#matrices como arreglos de arreglos
for x in A:
    print(x)
```

In [ ]:
```
# Podemos recorrer la matriz de la siguiente manera:
for x in A:
    for y in x:
        print(y)
```

In [ ]:
```
# otra forma, usando indices es:
for j in range(3):
    for i in range(2):
        print(A[i,j])
```

In [ ]:
```
# otra forma, usando indices es:
for i in range(2):
    for j in range(3):
        print(A[i,j])
```

In [ ]:
```
for i in range(2):
    print(A[i,1])
```

Extraer y utilizar partes de un arreglo:

In [ ]:
```
import matplotlib.pyplot as plt
import matplotlib
%matplotlib inline
from PIL import Image
```

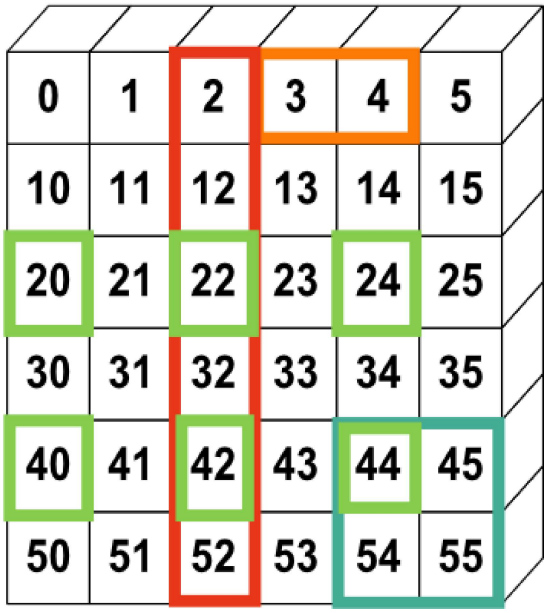
Existe una notación para extraer pedazos de arreglos en numpy que se conoce como indexado o "slicing"

```
>>> a[0,3:5]
array([3,4])


>>> a[4:,4:]
array([[44, 45],
       [54, 55]])

>>> a[:,2]
array([2,12,22,32,42,52])

>>> a[2::2,::2]
array([[20,22,24]
       [40,42,44]])
```



In [ ]:
```
#Creamos una matriz de 2x3:
A = np.array([[1, 2, 3], [3, 4, 5]])
print(A)
```

In [ ]:
```
#Ultima columna
A[:,2]
```

In [ ]:
```
A[:,:2]
```

Gráficos en Python

En ocasiones, debido a que ciertas ecuaciones diferenciales no poseen soluciones explícitas, es necesario realizar un análisis cualitativo de las soluciones, es decir, interpretar de manera gráfica algunos resultados. Por medio del estudio de la gráfica de la familia de soluciones podemos entender el comportamiento de las soluciones y poder usarlas para describir fenómenos, sin necesidad de utilizar alguna técnica de resolución para determinar la solución general de la EDO.

Para graficar funciones de una variable, utilizamos la librería `matplotlib.pyplot` cuya documentación completa se puede encontrar en el enlace

- https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.html (https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.html)

Para realizar un gráfico en 2D utilizamos la función `plt.plot()` , cuyos argumentos principales corresponden una lista o arreglo de valores asociados al dominio de la función y la función a graficar.

In [ ]:
```
import matplotlib.pyplot as plt
%matplotlib inline
```

Podemos hacerlo de esta forma:

In [ ]:
```
x=np.arange(-3.14,3.14,0.1)
y=np.sin(x)

plt.plot(x,y)  # Lineas por defecto, se puede usar "x" o "o" entre otros.
```

O de esta otra forma:

In [ ]:
```
x = np.linspace(0,20,100)
y = np.sin(x)

plt.plot(x,y)
```

Notamos que mientras la partición en x sea más fina, la curva es cada vez mas suave.

Es posible graficar más de una función en un mismo gráfico:

In [ ]:
```
from math import pi # para que reconozca la constante pi

x=np.linspace(-2*pi,2*pi,100)
y=np.cos(x)
f=np.exp(-x**2)

plt.plot(x,y)
plt.plot(x,f)
```

La presentación de las curvas en cuanto a color, grosor, forma, etc son presentados siempre por defecto de la misma manera. Si se desea personalizar el gráfico es posible utlizar algunas de las siguientes funciones:

- `linewidth` : Personaliza el grosor de la línea.
- `linestyle` : Personaliza el estilo de línea. El formato debe ser ingresado en cremillas, algunos formatos soportados son

```
* '-' o 'solid': formato por defecto.
* '--' o 'dashed': formato por líneas.
* '-.' o 'dashdot': formato línea punto.
* ' ' o 'none': formato ¿absurdo?
* ':' o 'dotted': formato de puntos.
```

- `color` : Personaliza el color de la curva en formato (R,G,B) o bien con nombre directo. Una buena página para seleccionar colores es https://matplotlib.org/stable/gallery/color/named_colors.html (https://matplotlib.org/stable/gallery/color/named_colors.html)

In [ ]:
```
import matplotlib.pyplot as plt
import sympy as sp #agregamos SymPy
%matplotlib inline

x = np.linspace(0,10,50)
y = np.sin(3*x)

plt.xlabel(sp.latex('variable independiente')) #etiqueta eje x
plt.ylabel(sp.latex('velocidad del auto')) #etiqueta eje y
plt.plot(x, y,linewidth=3, linestyle=':', color=(0.4, 0.1 ,0.9), label='función s
plt.grid()
plt.legend()
plt.show()
```

In [ ]:
```
plt.plot(x,y, linewidth=4, linestyle=':', color=(1, 0.5 ,0.5))
plt.plot(x,f, linewidth=2, linestyle='dashdot', color='dodgerblue')
```

En adición, es posible utilizar las siguientes funciones:

- `plt.xlabel(' ')` : Añade etiqueta al eje de variables independientes.

- `plt.ylabel(' ')` : Añade etiqueta al eje de varibles dependientes.
- `plt.grid()` : Agrega una grilla (o red) al gráfico.
- `plt.title('<titulo>')` : Agrega un título al gráfico.
- `plt.xlim([<lim inferior>, <lim superior>])` : Margen de visualización variable independiente
- `plt.ylim([<lim inferior>, <lim superior>])` : Margen de visualización variable dependiente
- `plt.legend()` : Agrega una leyenda al gráfico. Para ello es necesario agregar con anterioridad una etiqueta a cada función, vía `label` , por ejemplo `plt.plot(x,y, label='función')`

Ejemplo 1

Graficar las funciones $f(t) = t\,\sin\left(\frac{1}{t^2+1}\right)$ y $g(t) = \ln(t^2+1)$ con las siguientes consideraciones:

- f con grosor 3, línea continua, en color azul (a elección), etiqueta "f(t)".
- g con grosor 4, línea punteada, en color naranjo (a elección), etiqueta "ln(t^2)".
- gráfico de las funciones sobre la región $[0, 2\pi] \times [-2, 2]$
- gráfico con grilla, título "Mi primer gráfico" y con cuadro de leyendas.
- Etiquetas t e y respectivamente.

In [ ]:
```
t=np.linspace(0, 2*pi, 200)
f=t*np.sin((1/(t**2+1)))
g=np.log(t**2+1)

plt.plot(t, f, linewidth=3, color='aqua', label='f(t)')
plt.plot(t, g, linewidth=4, linestyle=':', color='darkorange', label='ln(t^2)')

plt.xlim([0,2*pi])
plt.ylim([-2,2])

plt.grid()
plt.title('Mi primer gráfico')
plt.xlabel('t')
plt.ylabel('y')
plt.legend()
```

Ejercicios:

a) Defina un arreglo de dos filas y dos columnas.

In [ ]:
```

```

b) Calcule la suma de los elementos de su arreglo.

In [ ]:

c) Calcule el promedio de los elementos de su arreglo.

In [ ]:

d) Determine la cantidad de elementos de su arreglo que son menores que 6.

In [ ]:

e) Determine el valor y posición del elemento más grande de su arreglo.

In [ ]:

f) Determine el valor y posición del elemento más pequeño de su arreglo.

In [ ]:

g) Determine el elemento que más se repite en su arreglo.

In [ ]:

Para los siguientes ejercicios haremos uso de una matriz aleatoria:

In [ ]:
```
# comando: np.random.randint(maxValor, size=(rows, columns))
np.random.seed(1)
B = np.random.randint(100, size=(10, 10))
B
```

h) Calcule la suma de los elementos de B

In [ ]:

i) Calcule la suma de elementos de la fila 3

In [ ]:

j) Calcule la suma de elementos de la columna 3

In [ ]:

k) ¿Cuál es la fila con la mayor suma? ¿Cuál es la coumna con la mayor suma?

In [ ]:

l) Diremos que una matriz es mágica si la suma los elementos de sus columnas y las filas es el mismo. Escriba un programa que reciba una matriz de 10 por 10 y retorne True si es mágica y False en otro caso.

In [ ]:

m) Diremos que una matriz es antimágica si la suma los elementos de sus columnas y las filas son todas distintas. Escriba un programa que reciba una matriz de 10 por 10 y retorne True si es antimágica y False en otro caso.

In [ ]:

n) Diremos que una matriz es diagonal si para todo $i \neq j$ el elemento en la posición (i,j) es cero. Escriba un programa que reciba una matriz de 10 por 10 y retorne True si es diagonal y False en otro caso.

In [ ]:

o) Diremos que una matriz es densa si al menos un 85% de sus coeficientes es distinto de 0. Escriba un programa que reciba una matriz de 10 por 10 y retorne True si es densa y False en otro caso.

In [ ]:

p) Graficar las funciones $f(t) = t^3 \cos\left(\frac{t}{t^2-1}\right)$ y $g(t) = e^{t+3}$ con las siguientes consideraciones:

- f con grosor 8, línea continua, en color verde (a elección), etiqueta "f(t)".
- g con grosor 6, línea punteada, en color rojo (a elección), etiqueta "g(t)".
- gráfico de las funciones sobre la región $[-3\pi, 3\pi] \times [-6, 6]$
- gráfico con grilla, título "Mi segundo gráfico" y con cuadro de leyendas.
- Etiquetas t e y respectivamente.

In [ ]: