Introducción a Python

Temario de la Clase 4

Numpy

- vectores y matrices $\rightarrow$ arreglos-arrays
- Operaciones con arrays
- Slicing etc.
- Interminables for vs operaciones con arrays.
- Expresiones lógicas con arrays • Máscaras - arrays lógicos.
- Ordenando elementos.
- Broadcasting
- ufuncs personalizadas
- cython y f2py

Listas para manejo de funciones matemáticas

Supongamos que queremos trabajar con los puntos de la función $f(x) = x * sin(x^2)$ en el intervalo $[0, \sqrt{2\pi}]$. Resolución de 100 puntos.

```
import math as m
n=100
dx=m.sqrt(2*m.pi)/(n-1.0)
x=[] #crea una lista vacia
y=[]
for i in range(n):
    x1=i*dx
    x.append(x1)
    y1=funcion(x1)
    y.append(y1)
```

Listas para manejo de funciones matemáticas

La función viene dada por:

```
def funcion(x):
    f=x*m.sin(x**2)
    return f
```

Para generar listas en forma pythonica es:

```
x=[i*dx for i in range(n)]
y=[funcion(x1) for x1 in x]
```

Esta la posibilidad de tener lista de listas para representar a una matriz. Cada elemento de la lista madre es a su vez una lista:

```
m=20; n=10
y=[]
for j in range(m):
    x=[i*j**0.5*dx for i in range(n)]
    y.append(x)
y=[[i*j**0.5*dx for i in range(n)]  for j in range(m)]
```

```
>>> y[2][5]
35.35533905932738
>>> len(y[2][:])
10
```

NumPy: Arreglos

Para realizar cálculos científicos y trabajar con vectores, matrices, tensores, etc existe una librería específica: Numerical Python “numpy”.

```
import numpy  as np
```

Por convención llamamos siempre a la librería numpy como np.

La estructura básica de numpy son los “arrays” o arreglos que son tensores de cualquier rango, esto incluye a vectores y matrices.
Para convertir una lista en un array, se usa `np.array`:

```
ar=np.array([5.0,2.3,7.2])
```

```
ar=np.array(lista)
```

Por cuestiones de eficiencia $\rightarrow$ Los arrays tienen dimensiones fijas!.

Los arrays de NumPy son parte central de todas las herramientas de ciencia de datos en Python.

Generación de arrays

Para generar un array, con `np.zeros`, lo llenamos de 0s o lo dejamos vacio con `np.empty`.

```
ar=np.zeros(n)
```

```
ar=np.empty(n)
```

Estos generan un vector de n componentes.

Para generar una matriz de n filas por m columnas hacemos

```
Mtx=np.zeros([n,m])
```

No olvidar los corchetes o doble paréntesis (se interpreta como el primer argumento de la función `np.zeros`).

Para ver los tamaños de los arrays:

```
print('tama~no: ',Mtx.shape)
print('orden o dimension del array: ',Mtx.ndim)
print('longitud total: ',Mtx.size)
print('tama~no primera dim: ',Mtx.shape[0])
```

Tipo de arreglos (especiales de numpy): `A.dtype`

Acceso a los elementos de los arrays

Para acceder a los elementos de un array se hace de la misma manera que en las listas, si es un vector

```
a[5]; a[0:2]; a[3:]; a[2:-3]
```

Recordar que los negativos son para ir desde atras hacia adelante!
Si es una matriz se agregan las dimensiones con comas:

```
a[0,5]; a[3,5:7]; a[2,:]
```

Si tengo un arreglo de alto orden y quiero trabajar con algun orden/índice:

```
nbatch,nchan,npixels = 32, 16, 256
Arr=np.zeros((nbatch,nchan,npixels,npixels))
```

A que elemento accedo cuando hago `Arr[0]`?
Y cuando hago: `Arr[:,0,...]`?
Y cuando hago: `Arr[...,0]`?

Operaciones con matrices

Si queremos generar la matriz identidad:

```
identidad=np.eye(N); identidad2=np.eye(N,M)
```

Si queremos generar un vector o una matriz de unos:

```
unos=np.ones(N); unos2=np.ones(N,M)
unos3=np.ones_like(unos2)
```

Si queremos generar un array igual a uno que ya generamos:

```
zeros2=np.zeros_like(unos2)
```

Si queremos extraer la diagonal de una matriz

```
A.diagonal()
```

Si queremos la matriz transpuesta:

np.transpose(),.T

Mas general si quiero intercambiar ejes:

np.swapaxes(array,axis1,axis2)

```
>>> A=np.random.randn(32,64,12,12)
>>> B=A.swapaxes(2,1)
>>> B.shape
(32, 12, 64, 12)
```

Generación arrays aleatorios para pruebas

Muestras de una distribución normal de media 0 y desviación 1:

```
from numpy.random import randn
n=15; m=10
A=randn(n,m,n)
B=np.sin(A)
```

Muestras de una distribución normal de media 2 y desviación 4:

```
alpha=0.1
A=np.random.normal(2,4,[n,m,n])
B=np.exp(-alpha*A)
```

Para acortar, en algunos ejemplos de estas filminas se asume randn esta importada.

`seed` Genera una semilla para que luego haya repetición de números aleatorios.

Reproducibilidad de los experimentos!

`rand` Genera muestras de una distribucion uniforme (0,1)

`choice` Genera muestras aleatorias a partir de una poblacion

`permutation` Permuta aleatoriamente una secuencia (variable nueva)

`shuffle` Permuta aleatoriamente una secuencia in place

Por supuesto `np.random` tiene generación de muestras de otras distribuciones: gamma, chisquare, binomial, etc (ya lo verán en Estadística).

Operaciones con matrices

Multiplicación por un escalar:

```
mtx1=alpha*mtx2; mtx3=alpha*ones(N)
```

Multiplicación entre vectores o matrices:

```
mtx1=np.matmul(mtx2,mtx3); alpha=np.dot(v2,v3)
```

Una nueva forma de escribir el producto matricial en python 3 es:

```
mtx1=mtx2 @ mtx3
```

Esta última es muchísimo mas clara.

Recuerden que para producir productos matriciales se debe cumplir que
`mtx1.shape=[n,m],mtx2.shape=[m,k]`

Que sucede si hay dimensiones extras en los arrays? Pueden explicar cuando se pueden hacer los productos matriciales?
`np.einsum` realiza un producto siguiendo la convención de Einstein:

```
np.einsum('ij,kij->kj',A,B)
np.einsum('ijkl,mjkl->imk',A,B)
```

Los argumentos de entrada de las funciones np son arrays

Evaluación de funciones matemáticas que dependen de arrays:

```
v2=np.exp(-v1); v3=np.log(v1); v4=np.sin(v1)
```

```
import numpy as np
x = np.arange(10)
y = np.sin(x)

X = np.array([np.arange(10),
              2*np.arange(10)])
Y = np.exp(-X)
Z = np.sqrt(X+Y)
```

```
>>> Z.shape
(2, 10)
```

Es totalmente transparente.

Evitamos los tediosos for anidados para evaluar a matrices o arrays de órdenes superiores.

Funciones de numpy vs fors

```
import numpy as np
import time
n=300
A=np.random.normal(0,1,[n,n,n])
B=np.zeros([n,n,n])

t0=time.time()
for i in range(n):
    for j in range(n):
        for k in range(n):
            B[i,j,k]=np.exp(A[i,j,k])
print('Termino el for en: ',time.time()-t0)

t0=time.time()
B=np.exp(A)
print('Termino la eval matricial en: ',time.time()-t0)
```

```
Termino el for en:  21.321789026260376
Termino la eval matricial en:  0.12111473083496094
```

numpy es mucho mas eficiente que realizar evaluaciones componente a componente con for's Siempre se debe intentar trabajar con vectores y matrices.

Generación de vectores uniformes

Cuando hacemos una tabla para graficación en general necesitamos generar puntos equiespaciados entre un valor mínimo y un máximo.
La función np.linspace nos genera el vector automaticamente:

```
xvec= np.linspace(xmin,xmax,1000)
```

Esto nos genera un vector que comienza con el valor xmin y determina con el valor xmax y tiene 1000 puntos.
La resolución que hay entre puntos es de:

$$dx = (xmax - xmin)/(nptos - 1)$$

Entonces si hacemos:

```
xvec=[xmin+i*dx for i in range(nptos)]
```

Cual es la diferencia?
En numpy tenemos la instrucción: `np.arange(1.5,10,2.3)` similar a `range` pero se puede utilizar con flotantes.

Concatenación y subdivisión (splitting) de arrays

a) Reshape. Cambio la forma de un array: `np.reshape(array,new_shape)`

```
>>> a=np.random.randn(16,3,3)
>>> b=a.reshape((4,4,3,-1))
```

b) concatenate

`np.concatenate(lista,axis=0)`

Si quiero juntar dos o varios arrays en uno solo a lo largo de un eje ya existente:

```
>>>  a=np.random.randn(10,15,15)
>>>  b=np.random.randn(1,15,15)
>>>  c=np.concatenate([a,b])
(11, 15, 15)
```

c) `np.stack(lista,axis=0)`

Si quiero juntar dos o varios arrays en uno solo a lo largo de un nuevo eje:

```
>>>  a=np.random.randn(15,15)
>>>  b=np.random.randn(15,15)
>>>  c=np.stack([a,b])
>>>  c.shape
(2, 15, 15)
>>>  d=np.stack([a,b],axis=2)
>>>  d.shape
(15, 15,2)
```

d) split Si quiero subdividir un arreglo en partes:

`np.split(array,`
`nro_de_subarrays, axis=0)`

```
>>>  a=np.random.randn(20,15,15)
>>>  a1,a2=np.split(a,2)
>>>  a1.shape
(10, 15, 15)
>>>  a=np.random.randn(15,15,20)
>>>  a_list=np.split(a,4,-1)
```

Operaciones con las componentes

Operaciones de suma y producto que podemos realizar con las componentes de un array:

```
a = np.array([2, 4, 3], float)
a.sum()
a.prod()
```

Alternativa:

```
np.sum(a)
np.prod(a)
```

Máximo o mínimo y sus ubicaciones

Para obtener el valor máximo o mínimo de los elementos de un array:

```
a.min()
a.max()
```

Si en cambio queremos determinar los índices del elemento donde se encuentra el máximo o mínimo se debe hacer:

```
ind_mn = a.argmin()
ind_mx = a.argmax()
```

Puedo realizar la operación a lo largo de un eje específico uso: `a.sum(1);a.min(2)` etc.

```
>>> a=np.random.randn(64,16,32,32)
>>> b=a.sum(1)
>>> b.shape
>>> (64, 32, 32)
```

Si quiero ver las numerosas funciones/métodos que tienen los arrays hacer:

```
dir(a)
```

Operaciones estadísticas

```
a = np.array([2, 4, 3], float)
a.mean()
a.std()
a.var()
```

Ejercicio: Tenemos un array pesos donde se tienen todos los pesos medidos (E.g. `pesos=np.randon.normal(15,3,100)`) y se quiere sacar la media y el error de las mediciones.

Broadcasting/transmisión en arreglos

Vimos que todo tipo de operación binaria entre arreglos requiere que sean del mismo tamaño? No es cierto:

```
>>>  a=np.random.randn(4,15,15)
>>>  b=a+10
>>>  b.shape
(4, 15, 15)
```

```
>>>  a=np.random.randn(4,15,15)
>>>  b=a*10
>>>  b.shape
(4, 15, 15)
```

Que sucede?

La multiplicación escalar sigue la misma lógica.

La regla de transmisión: Dos arreglos son compatibles para transmisión si cada dimensión desde el final tienen la misma longitud o la longitud es 1. La transmisión se hace sobre las dimensiones faltantes o las que tienen longitud 1.

Esta laxitud puede ir un poco mas lejos, a ver:

```
>>>  a = randn(4,15,15)
>>>  b = randn(4,15)
>>>  c = a * b[:,None,:]
>>>  b.shape
(4, 15, 15)
```

```
>>> a = randn(4,15,15)
>>> b = randn(15,15)
>>> c = a + b
>>> d = randn(15)
>>> dd= a+d
```

Que sucede?

¿Cuando tengo que agregar la dimensión y cuando no?

Y si pruebo con transmision hacia ambos lados?

Operaciones lógicas con arrays

Comparación de dos arrays. Esto lo realiza elemento por elemento:

```
>>> a = np.array([1, 3, 0], float)
>>> b = np.array([0, 3, 2], float)
>>> c= a > b
>>> print(c)
array([ True, False, False], dtype=bool)
```

Igualdad:

```
>>> a == b
array([False, True, False], dtype=
                    bool)
```

Comparación con un valor:

```
>>> c= a > 2
>>> print(c)
array([ False, True, False], dtype=
                    bool)
```

Si quiero combinar dos arrays lógicos con los operadores lógicos hay que usar: $||$,$\&$

```
>>> c = a == b | a > 2
array([False, True, False], dtype=bool)
```

Recordar que los True los podemos pensar como 1's, luego que resulta de?:

```
>>> a=randn(100)
>>> nro= (a>0).sum()
```

```
>>> a=randn(100)
>>> nro=  np.count_nonzero(a>0)
```

Indexado fancy

En una clase previa tuvimos una pregunta, sobre como podíamos acceder a ciertos elementos a la vez de una lista. Para arreglos de numpy esto lo podemos hacer con indexado fancy:

```
>>> arr=randn(10)
>>> frst-last=arr[[0,-1]]
```

Uso una lista de los elementos que quiero seleccionar. Cuando tengo múltiples dimensiones:

```
>>> arr=randn(10,12,5)
>>> sub_arr=arr[[1,2],[2,3],:]
```

Combinemos los índices y hagamos broadcasting sobre ellos:

```
>>> arr=randn(10,12)
>>> rows=np.array([5,6])
>>> arr[rows[:, np.newaxis], [2,3]]
```

Lo puedo pensar como arr[idx,idy] donde los idx son los indices de los x y los idy son los índices de los y.

`np_ix_` construye una mesh a partir de los índices:

```
a[np.ix_([1,3],[2,5])]}# returns the array
                       #[[a[1,2] a[1,5]], [a[3,2] a[3,5]]]
```

Indexado lógico

A veces quiero realizar operaciones solo a ciertos elementos de un arreglo. Como los selecciono?

```
>>> a = np.array([1, 3, 0,-2])
>>> sqrta = np.sqrt(a[a>=0])
```

Que sucede si es un arreglo multidimensional?

`arr=np.random.randn(10,12,5)`

Puedo reemplazar los negativos por NaN:

```
arr=np.random.randn(10,12,5)
arr[arr<0]=np.nan
arr=np.sqrt(arr)
```

También puedo devolver los índices donde se cumple un condicional:

```
>>>  a = np.array([1, 3, 0,-2])
>>>  ind=np.where(a > 0)
```

Luego lo puedo usar para hacer evaluaciones de ese u otro array en esos índices. Pero notar que podría directamente trabajar con las matrices booleanas

Ejemplo de selección de elementos

Supongamos que queremos realizar la raiz cuadrada a los elementos positivos de un array.

```
  >>> a = np.array([[4, -1, 9]], float)
>>> a >= 0
array([[ True, False, True], dtype=bool)
>>> np.sqrt(a[a >= 0])
array([ 2., 3.])
```

Puedo guardar la operaci on lógica en un array lógico y luego utilizarlo multiples veces:

```
>>> sel = (a >= 0)
a = np.array([[4, -1, 9]], float)
>>> np.sqrt(a[sel])
array([ 2., 3.])
>>> np.log(a[sel])
array([1.38629436, 2.19722458])
```

Para todo el array

Si queremos saber si se satisface para cualquier elemento del array o para todo elemento del array. Se realiza primero la operación lógica y luego

```
  >>> c = np.array([ True, False, False], bool)
>>> any(c)
True
>>> all(c)
False
```

`any` con que uno solo de los elementos del array sea `True` el resultado será `True`
`all` solo cuando todos los resultados del array son `True` dará un `True`.

- El resultado de estas operaciones es una variable lógica single (no es un arreglo!).
- Por otro lado, los condicionales sobre arrays dan como resultado arreglos.

El `any` y `all` se suelen utilizar en los condicionales:

```
if (A>0).all(): B=np.sqrt(A)
```

Los condicionales sobre arrays NO se pueden usar en `if`'s excepto que se usen con .any u .all

Ufunc

Son todas las funciones numpy y scipy que se aplican a cada elemento del array: np.sin

Internamente estan “vectorizando” o contemplando todos los elementos del array para hacer la operacion.

unary ufuncs son las que tienen un solo argumento de entrada. np.sin, np.cos etc

binary ufuncs adicion, multiplicacion etc

Las ufuncs tienen un conjunto de métodos propios: reduce, out, accumulate.

Para grandes operaciones conviene directamente invocar el array de salida:

```
x = np.arange(5)
y = np.empty(5)
np.multiply(x, 10, out=y)
print(y)
```

```
x = np.arange(1, 6)
np.add.reduce(x)
15
```

Que esta haciendo?

Cython, f2py y C

Si queremos realizar operaciones que son extremadamente costosas? y si los loops terminan siendo obligatorios?

- Ejemplo 1: una optimización los valores en un paso vienen predefinidos por el paso anterior por lo que no podemos evitar el loop
- Ejemplo 2: integraciones temporales de sistemas dinámicos o modelos. Para llegar a un t es necesario ir paso a paso.

cython permite llamar a codigos C o C++ e ir y volver entre python y C. De esta maneras ciertas partes del codigo pueden estar en C y otras en python
De la misma manera f2py, es un wrapper que compila Fortran 77 and 90 y genera cabezales que luego permiten ir y volver entre python y fortran.
Se generan subrutinas de fortran, se las compila con f2py y luego pueden ser llamadas como si fueran funciones de python.

```
import mdl.l96mod as tfor
def integ2scl(self,xold):

    x,xss=tfor.l96.tinteg2scl(self.kt,xold,xssold,self.xpar,self.hint,self.css,
                                                self.bss,self.dt)
    return x
```

`l96mod.cpython-36m-x86_64-linux-gnu.so, l96mod.so`

Guardado de arrays

En formato ascii:

```
>>> a = np.array([1, 2, 3, 4])
>>> np.savetxt('test1.txt', a, fmt='%d')
>>> b = np.loadtxt('test1.txt', dtype=int)
```

En formato binario (recomendado):

```
>>> np.save('test3.npy', a)
>>> d = np.load('test3.npy')
```

La extensión *npy* es la que se utiliza para datos binarios en python. Si uds no la agregan python la tomará por default.

En formato binario compactado, guardo varios arrays:

```
>>> np.savez('test3.npz', velocidad=v, posicion=x)
```

Para leer los arreglos luego tengo que cargar el archivo:

```
>>>  a=np.load('test3.npz') # solo se da cuenta que esta compactado
>>>  v=a['velocidad'] # accedo como si fuera un diccionario
>>>  x=a['posicion']
>>>  v2=a.f.velocidad  # alternativa para acceder a la velocidad
```

En que línea esta leyendo en el disco a la posición?