[*Escuela de código*]

Apuntes de clase:

Numpy y comprensión de listas

Tabla de contenido

Numpy

Numpy es una librería open source creada en 2005 utilizada en la mayoría de los campos de la ciencia, cálculo numérico e ingeniería. Es una librería de cálculo estándar que se utiliza en la mayoría de las librerías conocidas como scikit-learn, Keras, etc.

Se habla de que utilizar Numpy array en vez de el objeto "list" de Python para usos de un único tipo de dato (por ejemplo, una secuencia de números) es 50 veces más rápido utilizar Numpy que una lista.

Además, se habla de que puede llegar a ser hasta 2000 veces más rápido cuando se utiliza cálculo vectorizados con Numpy en vez de bucles.

Para instalar Numpy en nuestro sistema debemos tener "pip" instalado y ejecutar:

```
pip3 install numpy
```

Cuando queramos invocar a "numpy" en nuestro programa lo realizaremos de la siguiente forma:

```
import numpy as np
```

En general siempre verán en todos los programas de terceros que a "numpy" se lo importa con el alias "np", tal como figura en la documentación oficial.
A pesar de que esta es solo una "buena práctica" no regulada, será mal interpretado por terceros sino importamos a numpy con el alias "np".

Está librería está creada en C/C++, por lo que está optimizada. Otro punto importante a tener en cuenta es que los elementos "array" que se crean para almacenar los datos utilizan memoria contigua ya que al momento de crearlos se especifica tu tamaño y contenido. Estos dos puntos mencionados son los más relevantes en cuanto a porqué Numpy es tan performante y tan veloz.

Crear un array

Un array se genera utilizando el método "array". El array se debe completar con la información deseada al momento de generarlo, ya que es de tamaño fijo (se puede crear un array vacío o cambiar el tamaño luego pero es muy poco performante).

```
v1 = np.array([1, 2, 3])
```

Cuando definimos un array de una sola dimensión como el ejemplo anterior se dibuja como si fuera una columna. Además, se puede especificar el tipo de datos que deseamos que contenga el array, especificando así su precisión numérica. En caso de no declarar el tipo de dato Numpy elegirá el tipo de dato que crea más adecuado.

```
v1 = np.array([1, 2, 3], dtype=np.int64)
```

Tipo de dato "int" 64bits (precisión de 1.84 x 10^19)

Operaciones básicas

Para los ejemplos a continuación utilizaremos los siguientes dos array:

```
v1 = np.array([1, 2])
v2 = np.array([2, 2])
```

Suma

La suma de dos array es la suma de cada elemento de un array con su correspondiente en el otro (posición a posición):

```
suma = v1 + v2
print(suma)
# suma = [1, 2] + [2, 2] = [1+2, 2+2] = [3, 4]
# suma = [3 4]
```

Como pueden ver en una sola línea de código "**suma = v1 + v2**" me computa la suma de los arrays. Si quisiera resolver este problema con listas tendría que involucrar varias líneas de código y un bucle:

```
l1 = [1, 2]
l2 = [2, 2]
lista_suma = []
for i in range(len(l1)):
    suma = l1[i] + l2[i]
    lista_suma.append(suma)

print(lista_suma) # lista_suma = [3 4]
```

Este ejemplo nos muestra porqué es mucho más eficiente Numpy.

Producto por un escalar (un número)

El producto de un escalar (una constante, un número) por un array es el siguiente:

```
producto_escalar = 3 * v1
print(producto_escalar)
# producto = 3 * [1, 2] = [3*1, 3*2] = [3, 6]
# producto = [3 6]
```

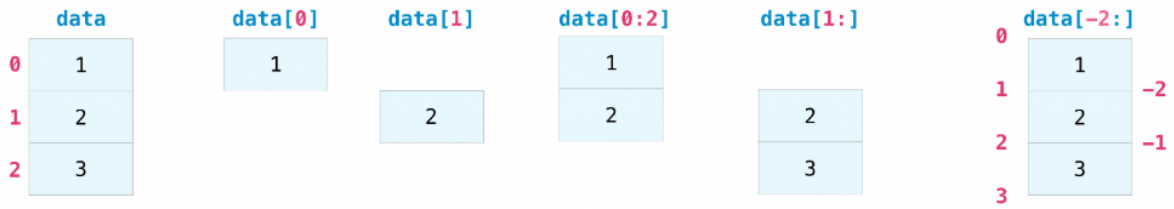
Producto vectorial

El producto entre dos array es el siguiente:

```
producto = v1 * v2
print(producto)
# producto = [1, 2] * [2, 2] = [1*2, 2*2] = [2, 4]
# producto = [2 4]
```

Slicing

El operador ":" funciona exactamente igual que en las listas y tuplas, pudiendo permitirnos acceder a ciertas porciones del array:



Atributos destacados

Los atributos más destacados de los objetos “array” son:

- ***shape:*** Retorna cuantas filas y columnas posee el array.
- ***size:*** Retorna la cantidad de elementos en el array.
- **ndim:** Retorna la cantidad de dimensiones del array(1D, 2D, 3D).

```
m = np.array([[1, 2], [3, 4], [5, 6]])
m = np.array([
                [1, 2],
                [3, 4],
                [5, 6]
                ])

print(m.shape)   # (3, 2) (fila, columna)
print(m.size)    # size=6 (3x2)
print(m.ndim)    # ndim=2 (array 2D)
```

Métodos tradicionales

Método de "**sum**" utilizando Numpy vs listas

```
top = 5000000

v1 = np.arange(top)
sum_v1 = np.sum(v1)

l1 = list(range(top))
sum_l1 = sum(l1)
```

```
Numpy time: 2.99ms
List time: 150.63ms
SUM: Numpy vs List x50.34
```

Tal como adelantamos al comienzo del apunte, para casos simples como la suma de todos los elementos de una secuencia de números Numpy completa la misma operación **50 veces** más rápido.

Método de "**mean**" utilizando Numpy vs listas

```
v1 = np.arange(top)
mean_v1 = np.mean(v1)

l1 = list(range(top))
mean_l1 = sum(l1) / len(l1)
```

```
Numpy time: 4.99ms
List time: 155.62ms
Mean: Numpy vs List x31.21
```

Método de "**sort**" utilizando Numpy vs listas

```
v1 = np.arange(top)
sort_v1 = np.sort(v1)

l1 = list(range(top))
sort_l1 = l1.sort()
```

```
Numpy time: 55.82ms
List time: 52.84ms
SORT: Numpy vs List x0.95
```

Otros métodos que también hemos utilizado en el pasado que también implementa numpy:

Método	Descripción	Ejemplo
round	redondeo	np.round(5.6) # resultado = 6.0
floor	redondeo bajo	np.round(5.6) # resultado = 5.0
ceil	redondeo alto	np.round(5.6) # resultado = 6.0
pi	número pi	np.pi # resultado = 3.141592...
sign	valor signo	np.sign(5.6) # resultado = 1.0
sqrt	raíz cuadrada	np.sqrt(4) # resultado = 2
abs	valor absoluto	np.abs(-4) # resultado = 4

Consultar la sección de "Links de interés" al final del documento para encontrar la documentación oficial de estos y otros métodos

Métodos nuevos

asanyarray

Método utilizado para convertir una secuencia list o tupla en un array Numpy.

where

Método utilizado para reemplazar valores en un array/lista en base a una condición:

```
l1 = list(range(top))
v1 = np.asanyarray(l1)

where_v1 = np.where((v1 % 2) == 0, v1, 0)
# [0 0 2 0 4 0 6 0 8]
```

```
np.where(cond, verdadero, falso)
```

Utilizar el método "where" en vez del clásico loop nos permite realizar hasta 10 veces más rápido el proceso.

diff

Método utilizado para calcular la diferencia entre cada valor del array y su siguiente. Por ejemplo:

```
v1 = np.array([1,2,4,7])
v1_diff = np.diff(v1)
# [2-1, 4-2, 7-4]
# [1 2 3]
```

¿Qué pasa si tenemos un array de dos dimensiones (2D), con filas y columnas? Por ejemplo:

```
m = np.array([
            [1, 5],
            [3, 1],
            [5, 10]
            ])
```

Si utilizamos el método "diff" el sistema por defecto nos devolverá la diferencia calculada fila a fila:

```
m_diff_por_fila = np.diff(m)
#[[ 4]
# [-2]
# [ 5]]
```

Pero especificando el "**axis**" podemos indicarle que queremos que devuelva el resultado columna a columna:

```
m_diff_por_columna = np.diff(m, axis=0)
# [[ 2 -4]
#  [ 2  9]]
```

axis=0 → Recorre por columna
axis=1 → Recorre por fila

Mask (máscara)

Las máscaras en numpy se utilizan para filtrar/reducir un array según un criterio, en este caso, vamos a reducir un array numpy a solo sus números pares. Dado el array:

```
v1 = np.array([1, 2, 4, 7])
```

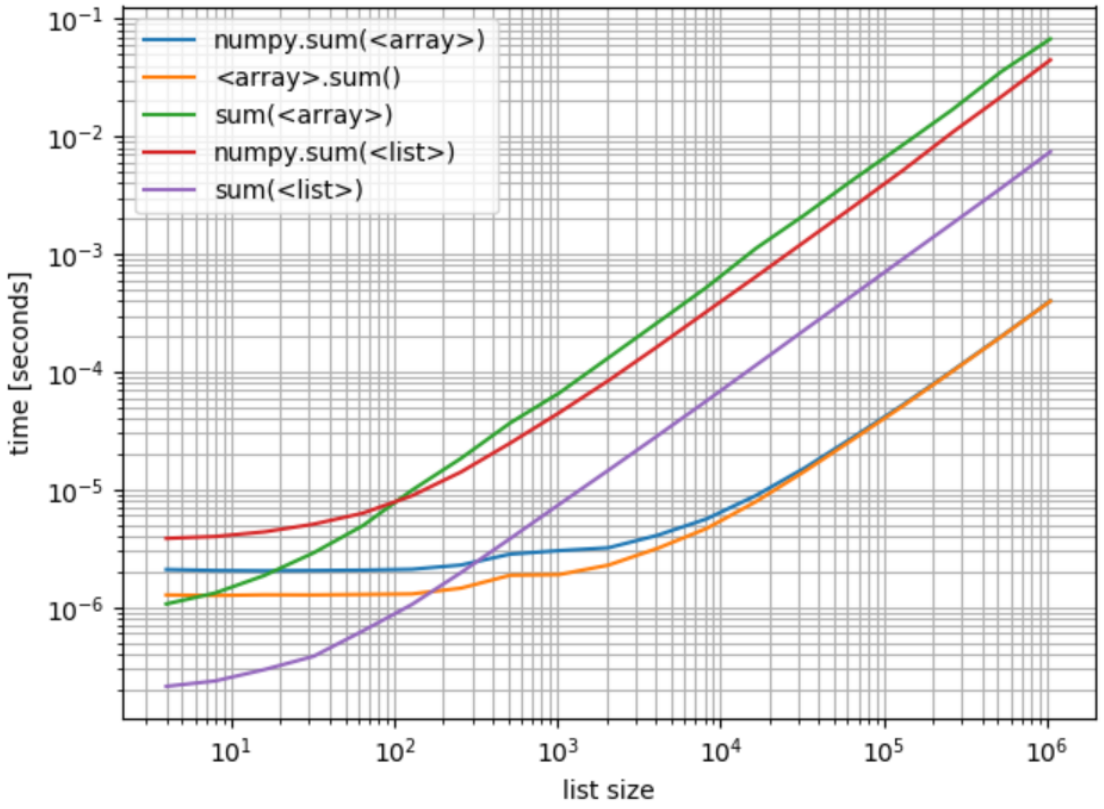
Crearemos una máscara con la condición que retorne "True" si el número es par:

```
mask_par = (v1 % 2) == 0
```

Dicha máscara la colocamos dentro del array "v1" para filtrar solo los valores deseados y obtener un array reducido nuevo:

```
v1_pares = v1[mask_par]
# [2 4]
```

Conclusión



Finalmente Numpy es grandioso y la performance que puede alcanzarse es muy superior pero hay un detalle **MUY IMPORTANTE** a tener en cuenta:

- Numpy es solo más performante si se utiliza arrays. En caso de utilizar los métodos Numpy con listas en la mayoría de los casos tendremos una degradación de la performance.
- A su vez se da la misma condición contraria, utilizar métodos nativos para listas en arrays Numpy produce una degradación de la performance. Esto se debe al hecho de necesitar transformar el tipo de dato para ejecutar la operación.
- Numpy es significativamente más performante a partir de los 1000 elementos, lo cual es un número fácilmente alcanzable.
- Para muy bajo niveles de datos es preferible utilizar los método tradicionales con listas.

Es importante tener estas consideraciones especialmente cuando utilizamos librerías de terceros que es probable que estén utilizando Numpy.

Lambda expression

Las lambda expression son en definitiva funciones "anónimas" declaradas en una línea de código. Su utilidad es hacer la sintaxis más clara y legible y poder declarar funciones "on the fly" para resolver un tema puntual que luego no será necesaria volver a utilizarla.

Una expresión lambda se forma de la siguiente forma:

```
lambda par_1, par_2..: par_1 + par_2
```

Al utilizar la sintaxis "lambda" el intérprete ya sabe que a continuación vendrá una función declarada en una línea.

- Lo primero que debemos declarar es la lista de parámetros de la función separados por coma.
- Luego se debe colocar el operador ":" (entonces) para declarar que comenzará la función.
- Luego se declara la operación que se realizará con los parámetros pasados a la función y se revuelve el resultado de dicha operación.

Por ejemplo queremos generar una función lambda que multiplique por "2" un valor ingresado, para ello tendríamos:

```
mult_by_2 = lambda x: 2*x
print(mult_by_2(3))  # resultado = 6
```

En este caso se está definiendo una función lambda que como parámetro se pasa un valor que dentro de la función llamaremos "x" el cual luego es multiplicado por "2" y retornado como resultado.

El equivalente a definir esta función de forma tradicional es:

```
def mult_by_2(x):
    return 2*x
```

Map

Map es un método de Python que nos permite "mapear" una función con una secuencia de datos. Map se encarga de aplicar la función deseada a toda la secuencia de datos pasada como parámetros.

Por ejemplo, deseamos mapear la función "abs" a la lista "numeros", generando así una iteración resultante en una nueva lista:

map(func, datos)

```
numeros = [1, -5, -6, 4]
numeros_abs = list(map(abs, numeros))
print(numeros_abs)
# [1, 5, 6, 4]
```

Esta nueva lista generada posee los valores de "numeros" sometidos a la función "abs".

Lo mismo se puede hacer con "map" y la función lambda "mult_by_2" que acabamos de generar:

map(lambda, datos)

```
numeros = [1, -5, -6, 4]
numeros_lambda = list(map(lambda x: 2*x, numeros))
print(numeros_lambda)
# [2, -10, -12, 8]
```

En este caso no estamos aprovechando que las funciones lambda pueden ser definidas en línea, para aprovechar todos los recursos vistos hasta ahora deberíamos escribir ese "map" como:

```
numeros = [1, -5, -6, 4]
numeros_abs = list(map(lambda x: 2*x, numeros))
print(numeros_abs)
# [2, -10, -12, 8]
```

Conclusión

Ya tenemos los conocimientos para implementar funciones lambda y aplicarlas a una secuencia de datos y generar nuevos a partir de dicha operación. Esta operación la podríamos haber realizado con un bucle como ya conocemos.
Sabemos que la ventaja de haber utilizado map + lambda expression es que la sintaxis es mucho más "pythonica" (legible, limpia, simple), pero ¿realmente es mejor que utilizar un bucle? Hagamos una análisis de performance:

Se recorrerá un rango de medio millón de elementos, a cada elemento se lo multiplicará por "2" y se lo almacenará en una lista.

Procedimiento con bucle

```
rango = range(500001)

# ----------------------------
# Método tradicional con bucle
lista_pares_bucle = []
for i in rango:
    valor = 2*i
    lista_pares_bucle.append(valor)
# ----------------------------
```

Procedimiento con lambda expression

```
numeros = list(rango)
# ----------------------------
# Método con map + lambda expression
lista_pares_lambda = list(map(lambda x: 2*x, numeros))
# ----------------------------
```

Resultado

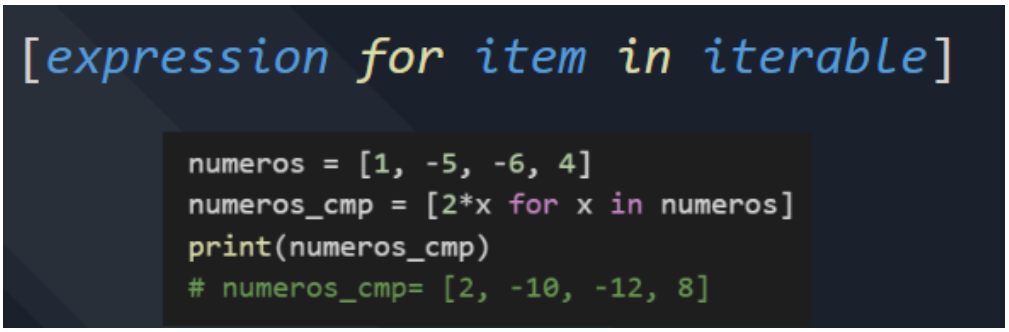
```
Bucle time: 76.79ms
List time: 79.79ms
WHERE: Lambda vs Bucle x0.96
```

Como podemos observar el hecho de utilizar "map + lambda expression" es levemente menos performante que bucle. La diferencia no es tal que debamos alarmarnos, pero hay que tenerlo en cuenta. Las ventajas mencionadas anteriormente superan la pérdida de performance.

Comprensión de listas

El concepto de comprensión de lista (o diccionarios, sets, etc) es exactamente el mismo que desarrollamos con "map + lambda expression" pero sin tener que agregar el "overhead" de del "map." Esto permite que sea aún más legible el código, más limpio y igual o más performante.

La sintaxis de una comprensión de lista básica es muy parecida a las ya vistas:



En el ejemplo dado estamos realizando exactamente la misma operación que antes logramos con "map + lambda expression" mucho más simplificado y ordenado.

Generador

Una de las mayores utilidades de la comprensión de lista es generar una nueva lista a partir de un rango o secuencia de datos. Esto se lo conoce como "generador". Algunos ejemplos:

Utilizando un rango de datos, en el cual se usa el valor del rango como parte del elemento calculado iteración a iteración que completa la lista

Procedimiento con bucle

```
lista = []
for x in range(10):
    valor = x**2
    lista.append(valor)
```

Procedimiento con comprensión

```
lista = [x**2 for x in range(10)]
```

Utilizando el rango d e datos únicamente para contar "x" cantidad de veces, en cada iteración realizar una determinada acción:

Procedimiento con bucle

```
lista = []
for x in range(5):
    valor = random.randint(1,6)
    lista.append(valor)
```

Procedimiento con comprensión

```
lista = [random.randint(1,6) for x in range(5)]
```

Filtro

Hay dos formas de "filtro" en cuanto a comprensión de lista:

- Una de ella consta de copiar algunos elementos de la lista origen y otros modificarlos.
- Otra forma consta solo de copiar algunos elementos de la lista y otros no, por lo cual la lista resultado será de menor tamaño.

En el **primer** caso la sintaxis se escribe como:

```
[out1 if cond else out2 for item in iterable]
```

Según si la sentencia condicional se cumple o no, la lista se tomará el valor de "out1" o "out2" en cada iteración. Se puede utilizar condicionales anidados o compuestos.

Ejemplo: Generar una lista que tome los números pares de "numeros" y reemplace los números impares por "0".

Procedimiento con bucle

```
lista = []
numeros = [1, 2, 4, 6, 8, 3 ,5]
for x in numeros:
    if(x % 2) == 0:
        valor = x
    else:
        valor = 0
    lista.append(valor)
```

Procedimiento con comprensión

```
numeros = [1, 2, 4, 6, 8, 3, 5]
lista = [x if(x % 2) == 0 else 0 for x in numeros]
# lista [0, 2, 4, 6, 8, 0, 0]
```

En el **segundo** caso la sintaxis se escribe como:

```
[out1 for item in iterable if cond]
```

Según si la sentencia condicional se cumple o no se realizará la iteración. Es por esto que si la condición nunca se cumple no se realizará ninguna iteración por lo que la lista resultante estará vacía.

Ejemplo: Generar una lista que solo tome los números pares de "numeros" y descarte los números impares por "0".

Procedimiento con bucle

```
numeros = [1, 2, 4, 6, 8, 3, 5]
lista = []
for x in numeros:
    if(x % 2) == 0:
        lista.append(x)
```

Procedimiento con comprensión

```
numeros = [1, 2, 4, 6, 8, 3, 5]
lista = [x for x in numeros if(x % 2) == 0]
# lista [2, 4, 6, 8]
```

Conclusión

Ya tenemos los conocimientos para implementar comprensión de listas. Es mucho más prolija y simple que implementar "map + lambda".
¿realmente es mejor que utilizar un bucle? Hagamos una análisis de performance:

```
rango = range(500001)

# ---------------------------
# Método tradicional con bucle
lista_pares_bucle = []
for i in rango:
    valor = 2*i
    lista_pares_bucle.append(valor)
# ---------------------------
```

```
# ---------------------------
# Método con comprensión
lista_pares_comp = [2*x for x in rango]
# ---------------------------
```

```
Bucle time: 70.84ms
Compresión time: 50.86ms
Compresion vs Bucle x1.39
```

¡En hora buena!

Tenemos una forma de mejorar la sintaxis, reducir la cantidad de código, realizarlo más limpio y que es igual o más performante que la forma clásica. Comprensión de listas es y será una de nuestras mejores herramientas de aquí en adelante.

Descargar un repositorio de GitHub

Para poder realizar las actividades de aquí en adelante debemos tener instalado y configurado nuestro GitHub. Todos los ejemplos prácticos estarán subidos al repositorio GitHub de **InoveAlumnos**, para aprender como descargar estos ejemplos desde el repositorio referirse al "Instructivo de GitHub: Descargar un repositorio" disponible entre los archivos del campus. De no encontrarse allí, por favor, tenga a bien comunicarse con alumnos@inove.com.ar para su solicitud.

Debemos descargar el repositorio que contiene los ejemplos de clase de ésta unidad:
https://github.com/InoveAlumnos/numpy_list_cmp_python

Hasta la próxima!

Con esto finaliza el tema "numpy y comprensión de listas", a partir de ahora tienen las herramientas para poder comenzar a comprimir su código!.

Si desean conocer más detalles sobre el contenido pueden iniciar un tema de discusión en el foro del campus, o visitar los "Links de interés" que se encuentra al final de este apunte.

Links de interés

- Numpy sitio oficial
- Tutorial Numpy oficial
- Numpy w3school
- Cuando usar list comprehension
- Lambda vs comprehension