Introducción a Python

Temario de la Clase 6

- Excepciones
- Decoradores
- Guardando y leyendo archivos json/pickle
- Guardando y leyendo HDF5 / Netcdf
- Scraping la red
- Estructuras de pandas
- Manejo de datos con pandas
- Guardado de datos con pandas

Manejo de errores

Cuando se produce algun error durante la ejecución Python detiene la ejecución y emite un mensaje de Error.

```
>>> 5./0.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: float division by zero
```

El mensaje nos da una traza del error dice el archivo (o los archivos) donde se produjo, la línea y el módulo.
Luego viene el tipo de error.

Control del error con una excepción

Si sabemos que algo puede dar error podemos `try` y si ocurre un error hacemos un `except`.

```
try:
    val=num/den
except ZeroDivisionError:
    print('Denominador nulo')
    val=np.nan
```

Debemos conocer el tipo de error que va a ocurrir.

Si quiero capturar cualquier error:

```
try:
    val=num/den
except:
    print('Cayo')
```

Ojo porque en este caso no va a funcionar el keyboard exception!.

Recomendada:

```
try:
    val=num/den
except Exception:
    print('Cayo')
```

Excepciones en el ingreso de datos

Para el problema con el ingreso de datos erroneos existe una solución sencilla con el uso de las excepciones.

```
def readInt():
    while True:
        val = input('Ingrese un valor entero: ')
    try:
        val = int(val)
        return val
    except ValueError:
        print (val, 'no es un entero')
```

Una forma mas general que me sirve para cualquier dato

Podemos hacer una funcion mas general que me defina el tipo de dato que se debe ingresar y que nos de el mensaje de error si no es del tipo correcto:

```
def readValue(valType,InputMsg,ErrorMsg):
    while True:
        val = input(InputMsg)
        try:
            val = valType(val)
            return val
        except ValueError:
            print (val, ErrorMsg)
```

Esta función puede ser usada para cualquier tipo de datos. Función polimórfica.

Control de los errores

Se puede generar una error voluntario en nuestro programa mediante la orden `raise`.

De esta manera podemos controlar cuando aparezcan errores que sabemos de antemano pueden aparecer.

Cuando le pasamos el `raise` le decimos el tipo de error y una cadena de carácteres que indique el contexto.

```
>>> if den==0:
...     raise ZeroDivisionError,"El denominador se hizo 0"
Traceback (most recent call last):
File "<stdio>", line 1, in <module>
ZeroDivisionError: El denominador se hizo 0
```

Programación funcional para hiperparámetros

Si requiero construir una función en los cuales se definen ciertos (hiper)argumentos de entrada y luego se llama con otros se puede realizar con funciones jerárquicas. La salida de la función externa es la función interna!

```
def fn_ext(n,k):
    alpha=0
    for i in range(n):
        alpha+=k**2
    print('alpha en fn_ext: ',alpha)
    def fn_int(y):
        print('alpha en fn_int: ',alpha)
        print('sumo:',alpha+y)
        return alpha+y
    return fn_int # saco a la funcion interna
fn=fn_ext(10,5) # Defino los hyper-argumentos
b=fn(15) # Ahora si uso la interna
c=fn(20) # muchas veces
```

De mucha utilidad para métodos que solo tienen un argumento de entrada

Decoradores

Un decorador es una función que recibe una función como argumento y devuelve una función como salida. Se usan para agregar una funcionalidad/feature a una función. Sin alterar la función original.

```
def decorador_nombre(fn):
    def decorada(*args):
        print( 'Estoy en funcion',fn.__name__ )
        salida = fn(*args) #llama a la funcion original
        return salida
    return decorada
```

Siempre voy a estar ingresando la función, llamando a la función con todos sus argumentos y retornando la función.
Para usarlos se utilizan dos formas:

```
>>> def prn(string):
>>>     print(string)
>>> decorador_nombre(prn)('arg
                    entrada')
Estoy en funcion prn
arg entrada
```

```
>>> @decorador_nombre
>>> def prn(string):
>>>     print(string)
>>> prn('arg entrada')
Estoy en funcion prn
arg entrada
```

Ejemplo de decoradores para usar numba

Compilador a tiempo real para optimizar códigos que contengan ciclos, funciones y arreglos de numpy.

Se usa a través de decoradores:

```
from numba import jit
@jit
def f(x, y):
    # A trivial example
    return x + y
```

Puedo paralelizar código con numba:

```
@numba.jit(nopython=True, parallel=True)
def normalize(x):
    ret = np.empty_like(x)
    for i in numba.prange(x.shape[0]):
        acc = 0.0
        for j in range(x.shape[1]):
            acc += x[i,j]**2
        norm = np.sqrt(acc)
        for j in range(x.shape[1]):
            ret[i,j] = x[i,j] / norm
    return ret
```

Numba con numpy:

```
from numba import jit
import numpy as np

x = np.arange(100).reshape(10, 10)

@jit(nopython=True) # o @njit
def go_fast(a): # Function is compiled to machine
                code when called the first
                 time
    trace = 0.0
    for i in range(a.shape[0]):    # loops
        trace += np.tanh(a[i, i]) # NumPy functions
    return a + trace                # NumPy
                                broadcasting

print(go_fast(x))
```

No funciona con estructuras de pandas.

Serialización

pickle and json son librerías para serialización de objetos/estructuras de python.

- Pickling/Encoding/Codificando es el proceso de convertir una estructura de objetos/(actually instancias!) en una secuencia de bytes/caracteres. (solo el estado no los métodos).
- Unpickling/Decoding/Descodificando es la inversa toma una secuencia de bytes/caracteries y la convierte en una estructura de objetos.

Mientras que JSON esta pensado como un formato de serialización de texto/ascii/uft8, pickle esta pensado como un formato binario/bytes.

Serialización con json

JSON (JavaScript Object Notation) es un formato de texto inspirado en JavaScript pero que es independiente del lenguaje, y a su vez es muy legible.
Tiene dos estructuras básicas: pares de keys y values (como los diccionarios), {'key':value} y listas ordenadas de valores (vectores) [val1,val2,val3]

```
>>> import json
>>> json.dumps(['foo', {'bar': ('baz', None, 1.0, 2)}])
'["foo", {"bar": ["baz", null, 1.0, 2]}]'
```

Puedo guardar un diccionario ordenando las claves:

```
>>> print(json.dumps({"c": 0, "b": 0, "a": 0}, sort_keys=True))
{"a": 0, "b": 0, "c": 0}
```

Puedo introducir espacios de tabulación:

```
json.dumps([1, 2, 3, {'4': 5, '6': 7}], separators=(',', ':'))
'[1,2,3,{"4":5,"6":7}]'
print(json.dumps({ '6': 7,'4': 5}, sort_keys=True, indent=4))
{
    "4": 5,
    "6": 7
}
```

Des-serialización de un json

```
import json
json.loads('["foo", {"bar":["baz", null, 1.0, 2]}]')
['foo', {'bar': ['baz', None, 1.0, 2]}]
```

Si lo que quiero es cargar un archivo json. Tengo que abrir el archivo y luego lo des-serealizo con `load`:

```
f = open('data.json')

# returns JSON object as
# a dictionary
data = json.load(f)

for i in data['diccio']:
    print(i)

f.close()
```

Archivo data.json:

```
{"diccio": [{"nombre": "Jose",
    "direccion": "Rivadavia 40"}
          {"nombre": "Domingo",
    "direccion": "Mitre 53"}]}
```

Serialización y des-serialización con pickle

```
import pickle
import numpy as np

data = [np.arange(8).reshape(2, 4), np.arange(10).reshape(2, 5)]

# Genero archivo mat.pkl y guardo  data
with open('mat.pkl', 'wb') as outfile:
    pickle.dump(data, outfile, pickle.HIGHEST_PROTOCOL)

# Abro archivo mat.pkl y leo
with open('mat.pkl', 'rb') as infile:
    result = pickle.load(infile)
```

`result` va a ser la lista de arrays que guarde.

¿Cuales son las diferencias, ventajas y desventajas de usar save de numpy a pickle?

Datos multidimensionales y de muy alta dimensionalidad

Supongamos que tenemos datos guardados de un array de dimension $(Nsamples, Nt, Nx, Ny, Nz)$ es decir un array/tensor de datos espacio-temporales.

Las desventajas/limitaciones cuando guardamos arrays ya sea en csv/pickle/numpy/binarios:

- Para sacar un solo dato se debe leer todo el archivo.
  Si tengo un archivo de $(Nsamples, Nt, Nx, Ny, Nz)$ y quiero leer la muestra 5.
- Ahora quiero guardar un tiempo mas en el array $(Nsamples, Nt + 1, Nx, Ny, Nz)$, tengo que borrar el archivo y escribir uno nuevo todo de vuelta!.
- Si el sistema donde lo guardo es distinto de donde lo quiero leer no tengo garantias de que conserve los datos. (e.g. little o big endian byte orders)
- Los formatos en ASCII, csv, json etc son extremadamente ineficientes en la longitud del archivo, el tiempo de escritura y el tiempo de escritura.

Formatos para subsanar estas deficiencias: HDF-5 o netcdf.

HDF-5: Hierarchical data format

Desarrollado por U.S. National Center for Supercomputing Applications

Dos tipos de estructuras:

- Un dataset es un array multidimensional
- Un grupo es un grupo de datasets y subgrupos (estructura jerárquica)

Se puede definir metadata para cada grupo o dataset con los atributos de los datos.

Para leer:

```
>>> import h5py
>>> h5f = h5py.File('data.h5', 'r')
>>> list(h5f.keys())
['temperatura']
>>> dset = h5f['temperatura'] # lee
                    temperatura!
>>> h5f.close()
```

Para crear dataset:

```
h5f = h5py.File('data.h5', 'w') #
                      'a' para
                     append
h5f.create_dataset('temperatura',
                       data=np.ones
                      ((10,10)))
h5f.close()
```

Scraping la web

Existen varios paquetes que permite buscar sitios web, bajar contenidos y parsearlos.

`urllib` esta dentro de las librerías estandards de python. URL=Uniform Resource Locator

Vamos a abrir el sitio de la diplo:

```
>>> from urllib.request import urlopen
>>> page = urlopen('https://exa.unne.edu.ar/diplomatura/index.php/plan-de-
                                        estudios/')
>>> page
  <http.client.HTTPResponse at 0x14f9a8234e20>
```

Ahora vamos a leer el contenido de la página y pasarlo desde byte a ascii:

```
>>> html_bytes = page.read()
>>> html = html_bytes.decode("utf-8")
>>> print(html[:100])
    <!DOCTYPE html>
    <html lang="es">
    <head itemscope itemtype="http://schema.org/WebSite">
```

Tenemos la página en un string de python!