Notas de Pyhton

Lucho Cervantes Jorge Luis

2 de agosto de 2024

1. Comentario inicial

No hay mejores notas que la documentación original, de este modo, el presente trabajo no busca sustituir a dicha documentación. Por el mismo motivo, la estructura y los conceptos no son tan rigurosos. Este trabajo es más un catálogo, con una muy breve y probablemente incompleta descripción, de las herramientas que utilizo frecuentemente. El presente trabajo tiene como base inicial el curso *Python desde cero* [1] y posteriormente se va complementando con más cursos que se van mencionando en las referencias.

Índice

2. Preeliminares y atajos

Se hacen con #. Para hacer comentarios de más de un renglón se usan las triples comillas:

```
# Esto es un comentario
""" Todo esto es un comentario"""

```

Identación: Se refiere a que todas las instrucciones deben estar alineadas a la izquierda. Las sub instrucciones deben estar un espacio a la derecha de la instrucción correspondiente.

Una vez iniciada su ejecución, el programa se interrumpe con ctrl+z.

Para copiar el código seleccionado en una línea por debajo se usa shift+Alt+flecha arriba/abajo.

Es equivalente escribir x=x+a y escribir x+=a (y no se limita a la operación suma).

En VS Code, el script se guarda con ctrl+s y se ejecuta con ctrl+alt+n.

Para buscar una palabra específica en el código se usa ctrl+f.

Para reemplazar una palabra específica en el código se usa ctrl+h.

Para escribir en más de una línea a la vez se usa ctrl+alt+ flecha abajo hasta abarcar todas las líneas en las que queramos escribir.

3. Clases

Son los tipos de objetos con los que se trabaja en Python. Algunos de ellos son:

- ***int*:** números enteros.
- ***float*:** flotantes o números reales.
- ***bool*:** valores booleanos o variables lógicas. Toman el valor True o False (deben escribirse iniciando con mayúscula).
- ***str*:** string. Es una cadena de texto, esta se coloca entre comillas.
- ***list*:** Listas argumentos (arg) con el formato

```
x = [arg1, ..., argn]
```

Los arg pueden ser de cualquier clase, incluso pueden ser otras listas. Sus arg se enumeran de izquierda a derecha comenzando con el 0.

* Para indicar el arg i-ésimo de una la lista x, se usa x[i].

* Para elegir un rango de arg se usa x[inicial : final : paso] donde el inicial se toma en cuenta y el final se omite. Si el paso es negativo. el barrido de la lista se invierte (i.e. es de derecha a izquierda).

- ***tuple*:** tupla con el formato

```
x = (arg1, ...,argn)
```

Los arg también pueden ser de cualquier clase. La única diferencia con las listas, es que las tuplas no pueden ser modificadas. i.e. no se les pueden aplicar métodos como .append() , .pop() etc.
Una tupla se puede obtener de una lisca como:

```
tupla = tuple([a0, ..., an])
```

Una tupla también se puede obtener directamente con:

```
tupla = a0, ..., an
```

Las tuplas son convenientes cuando contienen datos de solo lectura.

- ***dict*:** diccionarios. Son similares a las listas, con la salvedad de que los arg no están ordenados mediante índices enteros que se ordenan de manera creciente. Los indices son personalizados y se denominan claves. Tienen el formato

```
x = {'Clave1':arg1, ..., 'Claven':argn}
```

Nuevamente, los arg pueden pertenecer a cualquier clase, a diferencia de las claves. En particular se puede poner una tupla o un frozenset como clave. Para elegir un elemento se usa x['Clave'].

Un diccionario x también se puede generar con:

```
x = dict(clave1=a1, ..., claven=an)
```

Puede ocurrir que se quiera crear un diccionario con ciertas claves sin contar aún con los valores para dichas claves. Para esto se usa:

```
x = dict.fromkeys(['clav1', ... , 'clavn'])
x = dict.fromkeys('ABCD', 'previo')
```

En el primer caso, se obtiene un diccionario con las claves deseadas, a las cuales se les asignan valores "none". En el segundo caso, se crea un diccionario donde las claves son cada uno de los caracteres del primer arg (A, B, C, ...) a las cuales se les asigna el valor 'previo'.

- ***set***: conjuntos. Son de la forma:

```
x = {a, x, ..., r}
```

Es similar a una lista con la salvedad de que sus elementos no pueden repetirse y no tienen un orden (o no están indexados). Esto último implica que no está definido un i-ésimo elemento de x, por lo que x[i] no tiene sentido. Solo puede mostrarse el conjunto completo.

Se pueden generar a partir de una lista:

```
x = set([a0, ..., an])
```

Los elementos del conjunto deben ser inmutables, así una tupla puede ser un elemento de un conjunto, no así una lista un, directorio u otro conjunto. Para que un conjunto1 tenga como elemento a un conjunto2, este último debe ser con la función **frozenset()**.

Para borrar un objeto x, se usa **del** x.

Dado un objeto de la forma x=(a0,...an). La asignación:

```
x0 = x[0]
   .
   .
   .
xn = x[n]
```

es equivalente al "desempaquetamiento":

```
x0, ..., xn = x
```

4. Funciones básicas

A los objetos de las distintas clases se les pueden aplicar las siguientes funciones:

- +, -, *, /, % : operaciones aritméticas de suma, resta, multiplicación, división y módulo respectivamente.
- print() : muestra en pantalla su arg

  Si se quiere colocar el valor de una variable x dentro del string se usa:

```
print('texto %i texto' %x)
```

  Se usa %i, %s o %.nf si x es int, str o un float redondeado a n decimales, respectivamente.

  Si se tienen más variable se usa:

```
print('%i texto %s...' %(x1, ...,xn))
```

  manteniendo el orden de aparición. Una manera equivalente de esto es:

```
print(f'{x1} texto {xn}...')
```

- type () : muestra la clase al que pertenece el arg.
- dir() : muestra los métodos (Todo lo que se lo puede aplicar) a la clase del arg.
- help() : muestra un resumen con la información del arg.
- range() : genera un contador según el arg.

  * El arg es de la forma (inicio, fin, paso). El final se excluye.
- list() : genera una lista con el contador range que se coloque como arg.
- sum() : suma los elementos de la lista, directorio o tupla utilizada como arg.
- len() : muestra cuántos elemento tiene la lista, directorio o tupla puesta como arg.
- all() : devuelve False si el objeto que se pone como arg contiene al menos un valor False como elemento. De lo contrario devuelve True.
- filter(func, x): a partir de una lista x y de una función func que devuelve solo True o False, almacena los elementos de x en los cuales func devuelve True. El objeto donde almacena dichos elementos se puede convertir a una lista.

5. Métodos

Acciones o funciones que puede realizar un objeto. La diferencia con las funciones es que los métodos siempre van a estar asociados a un objeto en específico. Los métodos se aplican colocando un punto después del objeto, seguido del nombre del método.

Aunque basta aplicar la función dir() a un objeto para conocer todos los métodos que se le pueden aplicar, algunos métodos más frecuentes son:

5.1. Métodos de strings

- .strip(): quita el caractér que se coloque como arg a la cadena que se aplica.
- .split(): separa la cadena de caracteres a la que se aplica, en secciones determinadas por el caractér colocado como arg.
- .upper(): pasa el string a mayúsculas.
- .lower(): pasa el string a minúsculas.
- .capitalize(): pasa el string a minúsculas y luego la primera letra la pasa a mayúscula.
- .index(): busca la cadena de caracteres puesta como arg, en la cadena de caracteres a la que se aplica. Entrega la posición en donde empieza el arg, si este no se encuentra, marca un error.
- .find(): hace lo mismo que index, pero si no se encuentra la cadena devuelve -1.
- .isnumeric(): Toma valores **True** y **False** según si el string al que se aplica está compuesto de puros números.
- .isalpha(): Toma valores **True** y **False** según si el string al que se aplica está compuesto de puras letras del abecedario.
- .count(): cuenta cuántas veces aparece el string del arg en la cadena a la que se aplica
- .__len()__: cuenta cuántos caracteres tiene el string al que se aplica.
- .endswith(): toma valores **True** y **False** según si el string al que se aplica termina con el string del arg
- .startswith(): toma valores **True** y **False** según si el string al que se aplica inicia con el string del arg
- .replace(str1, str2): reemplaza en el string al que se aplica el string str1 por el str2 [2].

5.2. Métodos de listas

- .append(): agrega el arg la lista a la que se le aplique. Se agrega al final.
- .insert(indice, arg): agrega el arg en el índice especificado.
- .extend([x1, ..., xn]): agrega los elementos x1, ..., xn al final de la lista.
- .pop(): quita el elemento de la lista con el índice de posición colocado como arg y lo guarda.
- .remove(): elimina el elemento de la lista con el índice de posición colocado como arg.
- .clear(): elimina todos los elementos de la lista.
- .sort(reverse=): ordena los elementos de la lista de mayor a menor jerarquía por default o si reverse = True, y de menor a mayor si reverse = False.
- .reverse(): invierte el orden de los elementos de la lista.

5.3. Métodos de diccionarios

- .keys(): proporciona todas la claves del diccionario. Para iterar sobre el diccionario basta guardar las keys en una variable x y colocar for i in x:
- .values(): proporciona todos los elementos del diccionario.
- .get(): devuelve el valor de la clave puesta como arg. Si la clave no está en el diccionario se devuelve none, a diferencia de cuando se usa nombre_diccionario['clave'] donde se marca un error y se detiene el programa.
- .items(): entrega una lista donde cada elemento es una tupla. Cada tupla contiene una clave y su valor en el diccionario.

5.4. Métodos de conjuntos

- .issubset(): Entrega True o False según si el conjunto1 al que se le aplique es subconjunto del conjunto2 colocado como arg. Es equivalente usar conjunto1 <= conjunto2.
- .issuperset(): Entrega True o False según si el conjunto1 al que se le aplique contiene al conjunto2 colocado como arg (conjunto1 > conjunto2).
- .isdisjoint(): Entrega True o False según si los conjuntos en cuestión son disjuntos.

6. Input

Función que pide al usuario introducir un string con algún dato, valor o simplemente una interacción, para poder continuar ejecutando el código. Por ejemplo, si deseamos que el usuario defina el valor de una variable x se usaría:

```
x = int(input ('introduce el valor de x: '))

```

La ejecución se detendrá hasta que el usuario compute el valor deseado y de enter. S e usa str(), int(), float(), bool() para convertir los datos a texto, números enteros, flotantes o valor booleanos etc.

Nota: En VS Code el programa debe ejecutarse en una terminal y no en el bloque de salida. I.e. se debe seleccionar la opción "run file" en lugar de ctrl+alt+n (boton de play).

7. Flujo del programa

7.1. Ciclo for

Su estructura más básica es:

```
for i in range(n,m,p):
    instruccion

```

Repite la instrucción seguida de los dos puntos según i valla tomando los valores definidos por el *range*(). Este último se puede cambiar por una lista, directorio o tupla, de modo que el ciclo for se aplica con i barriendo los elementos de tales objetos.

La estructura de este ciclo no es única, por ejemplo, el ciclo:

```
x = []
for i in range(n,m,p):
    x.append(valor)

```

es equivalente a escribir:

```
x = [valor for i in range(n,m,p)]

```

Para recorrer a la vez m listas x1, ..., xm con la misma longitud se utiliza la función zip():

```
for i1, ..., im in zip(x1,..., xm):

```

En este caso i barre los elementos de x1 y j los de x2. Una forma útil de imprimir los elementos de una lista x con un ciclo for es colocando enumerate(x) en lugar de solo x:

```
for i in enumerate(x):
    print(i)

```

de esta forma se imprimen tuplas con el índice de posición y el valor de los elementos de la lista. Esta forma es importante pues permite iterar sobre un **set** ya que le proporciona índices a sus elementos.

Para iterar sobre un directorio x:

```
for i in x:

```

donde i barre únicamente las claves del directorio, para acceder a los valores de dichas claves, se usa:

```
for i in x.items():

```

en este caso i toma como valor a las tuplas de la forma ('calv1', valor1), ..., ('calvn', valorn).

7.1.1. Interrupción de un ciclo for

Para que un ciclo se detenga cuando se cumpla una condición se usa lo siguiente:

```
for i in range (n,m,p):
          .
          .
          .
    if condicion:
        break

```

Si lo que se quiere es que se salte a la siguiente iteración en lugar de que se detenga el codigo se usa:

```
for i in range (n,m,p):
          .
          .
          .
    if condicion:
        continue

```

En particular, una línea de caractéres se puede iterar, por ejemplo:

```
for i in range 'ABCDE':
    print(i)

```

imprime en un rengón diferente, la letra A, en otro la B etc.

7.2. Ciclo while

Tiene la siguiente estructura:

```
while condicion:
    instruccion

```

donde la instrucción colocada después de los dos puntos se lleva a cabo mientras se cumpla la condición establecida. Para interrumpir el programa se utiliza ctrl+c. Si la condición toma valor True, el programa se ejecuta, si toma el valor False, se detiene. Para establecer la condición se puede usar $<$, $>$, $>=$, $<=$, $==$ (equivalencia), != (distinto) etc. ; los operadores lógicos **and**[1], **or**[2], **not** y lo operadores de pertenencia **in** y **not in**.

7.3. if, elif y else

Para condicionar al código de manera que este se ejecute solo cuando se cumple unca condición especifica se usa:

```
if condicion:
    instruccion

```

si se cumple la condición se ejecuta la instrucción que se encuentra después de los dos puntos. Si se quiere agregar otras condiciones con sus respectivas instrucciones se usa:

```
elif condicion:
    instruccion

```

Si no se cumplen las condiciones establecidas en el if y todos los elif agregados, se usa:

```
else:
    instruccion

```

Para indicarle al programa qué hacer.

Dos sentencias útiles son **break**, que permite romper los ciclos while y for en un punto deseado con ayuda de una condición impuesta con un if.

y **pass**, que permite saltar a la siguiente iteración del ciclo aunque el ciclo en curso no se haya completado con ayuda de una condición impuesta con un if.

8. Función definida por el usuario

Sirve para facilitar la aplicación de un bloque de código pensado para aplicarse más de una vez y no necesariamente al un mismo objeto de la misma clase. Una función se define de la siguiente forma:

```
def nombre_func(arg1, ... , argn):

    """-----------------------------------
    Documentacion ...
    -----------------------------------"""

    Cuerpo de la funcion (bloque de codigo)
    donde se establece como obtener val1,...
    hasta valn, a partir de arg1, ..., argn.

    return val1, ..., valn

```

se toma todo el bloque de código seguido de los dos puntos y la documentación (que describe brevemente lo que hace la función, los requisitos para usarla, condiciones etc.) se resume en la función nombre_func(), de modo que todo el bloque de código se podrá aplicar a distintos objetos, especificados como arg, simplemente llamando a la función.

Los valores obtenidos como resultado de aplicar la función se devuelven como elementos de un arreglo, al colocar la palabra return seguido de los valores separados por comas. Esto permite trabajar con ellos. Así,

```
x = nombre_func(arg1=a1, ... , argn=an)

```

Es un arreglo, con los valores de la función evaluados en los valores a1, ..., an. I.e. es de la forma:

```
x = [val1(a1, ..., an), ..., valn(an, ..., an)]

```

[1]Se puede sustituir por &
[2]Se puede sustituir por |

Cuando se desea que el argumento de la función sea una lista x cuya longitud puede variar según la cantidad de elementos que el usuario desee operar, se usa:

```
def nombre_funcio n(*x):
    # Se manipula x como una lista
    f(x)

# Para llamar a la funcion:
nombre_funcion(a0, ..., an)

```

en este caso el conjunto de elementos a0, ..., an se convierte en una lista independientemente de la cantidad de elementos que se traten y luego se opera según el código de la función. Si además de esto parámetros se desean agregar otros parámetros ya definidos, entonces *x se coloca siempre después de estos.

9. Funciones lambda

Son funciones sencillas, de un solo parámetro x y que se crean de manera más compacta. Estas se usan cuando se quiere hacer más rápido el trabajo, pues se salta el nombramiento de la función y se evita la creación de un bloque de código. Estas son de la forma:

```
lambda x : f(x)

```

A las funciones lambda también se les conoce como funciones anónimas.

A las funciones lambda también se les conoce como funciones anónimas.

10. Control de errores

Cuando el código presenta algún error, la ejecución se interrumpe al llegar a la línea que contiene dicho error y las siguientes líneas de código no se ejecutan. Para tener control sobre esta situación se utilizan las palabras reservadas:

- **try:** Indica la parte del código que debe ejecutarse y que se considera pudiera contener algún error.
- **except:** Si la parte que se indicó con try no puede ejecutarse debido a un error, se ejecutará lo indicado después de esta palabra en lugar de que el programa se detenga por completo.
- **finally:** La instrucción que se indique después de esta palabra se ejecuta independientemente de si existen o no errores en el código.

11. Entrada y salida de datos en un archivo de texto

Para abrir un archivo de texto se crea primero el objeto (en este caso de nombre **file**) que lo contendrá y se abre utilizando la función open() de la siguiente manera:

```
file = open('nombre_archivo.txt', 'modo')

```

donde los modos pueden ser los siguientes:

- r : lectura (modo por default).
- w : escritura.
- x : creación exclusiva (si el archivo ya existe, no se hace nada).
- a : añadir (agrega nueva información al archivo en cuestión).
- b : lo abre en modo binario.
- t : texto.
- + : leer y escribir.

Al objeto del archivo se le pueden aplicar los métodos:

.read(i): Se leen los primeros i caracteres del archivo como una variable string, contando espacios y saltos de línea. Si no se coloca nada como arg, se lee todo el archivo. Por ejemplo:

```
data = file.read()

```

Es una variable string que contiene todos los caractéres del archivo. Por su puesto se le puede aplicar la función print() para ver el archivo.

Si se vuelve a aplicar la función read() al archivo, con un nuevo valor de i, la lectura continuará a partir de la posición hasta la que llegó i en la primera llamada de read(). Para corregir esto (iniciar la lectura desde el inicio), se debe colocar file.seek(0), donde el 0 indica el inicio del archivo, naturalmente este arg puede cambiarse a voluntad.

.close() : sirve para cerrar el documento en python. Se debe colocar siempre.

.readline() : se lee el archivo hasta encontrar un salto de línea. Análogamente a read(), al aplicarse otra vez la lectura comienza donde se quedó la lectura anterior.

.readlines() : se leen todas las líneas y cada una de ellas es colocada como un elemento de una lista. Cada línea incluye el salto de línea '\n', que aparecerá como strings en todas las entradas, siendo lo único en los casos donde las líneas sean líneas vacías. Para deshacernos de ellos a los elementos de la lista se les aplica el método strip (que sustrae el arg de la cadena de caractéres a la que se le aplique).:

```
data = file.readlines()
data[i] = data[i].strip('\n')

```

En este caso data es una lista donde cada elemento es una línea del archivo.

Por ejemplo:

```
# Se generan datos:
P = [1*2*i for i in range(100)]
T = [2*i/5 for i in range(100)]

# Lista cuyos elementos son
# las lineas del archivo
datos = []
for i in range(200):
    datos.append('%s,%s,%s\n' \%(i,P[i],T[i]))

# Se escribe el archivo
file = open('archivo.txt', 'w')

# Se escribe el archivo
for i in out:
    file.write(i)
file.close()

```

Una manera más compacta de escribir el archivo es como sigue:

```
with open('archivo.txt', 'w') as file:
    file.writelines(datos)

```

En este caso, la palabra reservada **with** abre el archivo como "file" pero solo durante la ejecución de las líneas de código contenidas en el bloque identado (por lo que ya no es necesario file.close()). Por otro lado, el método .writelines() sustituye la combinación ciclo for - método .write().

tanto con **open** como con **whit open as** puede ocurrir que el archivo tenga caractéres que no se lean correctamente, para esto, en el caso del idioma español, se puede agregar el argumento encoding='UTF-8'

12. Módulos

Un módulo es un conjunto de declaraciones ejecutables y de definiciones de funciones ya establecidas de manera externa, las cuales pueden importarse al script de trabajo que se esté desarrollando y usarse solo mediante una referencia, sin tener que copiar el código de dichas funciones y sentencias [3].

Para usar un módulo, al inicio del código se tienen que llamar de la siguiente manera:

```
import nombre_modulo as nombre_abreviado

```

Para poder utilizar las funciónes que acompañan a dichos módulos se tienen que llamar de la siguiente manera:

```
nombre_abreviado.funcion()

```

Si el módulo o librería a importar es muy grande, al grado de afectar el rendimiento del ordenador, y solo se quiere utilizar unas cuantas funciones de todas las que se ofrecen, Estas se pueden importar de la siguiente manera:

```
from modulo import funcion1, ..., funcionn

```

En particular:

```
from modulo import *

```

Importa todas las funciones del modulo.

Algunos de los módulos más utilizados son **math, cmath, time, random, numpy, matplotlib, pandas, skyimage, sympy, scipy, json, csv, tensorflow** entre muchos otros. Algunos de estos son externos y deben instalarse primero. Para esto primero se corrobora que en efecto no se cuenta con tales módulos, esto se hace en una terminal con el comando:

```
python -m pip freeze
```

El cual muestra con cuáles módulos se cuenta.

Para instalar un módulo, también el la términal y con conexión a internet, se usa el comando

```
python -m pip install nombre_modulo
```

12.1. Módulo time

Nos permite trabajar con el tiempo utilizando el reloj de la computadora. Algunas de sus funciones son:

- time(): brinda el tiempo en segundos, Contando a partir de la fecha dada por time.gmtime(0)
- time_ns(): hace lo mismo que time pero en nanosegundos.
- perf_counter(): indica cuánto tiempo ha pasado entre la llamada de esta función y la llamada anterior.
- sleep(): al llegar a esta linea de código el programa se detiene la cantidad de tiempo indicada como arg.
- strftime('%c'): entrga la fecha en un formato entendible para el humnano.

12.2. Módulo json

Permite codificar un objeto de python en un archivo para poder recuperarlo en un futuro. Esto lo hace mediante la función dumb() que guarda un objeto en un archivo de texto. Por ejemplo, para guardar una lista en un archivo de texto se tiene el siguiente código:

```
import json

lista = ['%.2f' %(i*0.5) for i in range (20)]

with open('nombre_archivo', 'w') as file:
    json.dump(lista, file)

```

Para invocar dicha lista en un escript completamente diferente:

```
import json

with open('nombre_archivo', 'r') as file:
    lista=json.load(file)

```

12.3. Módulo numpy

Usualmente se importa como np y sirve para crear nuevos objetos conocidos como arrays (arreglos) que, a diferencia de las listas inherentes de pyhton, tienen definidas funciones que se llaman de igual forma que cuando se llama una función a una sola variable, pero que en realidad operan, a la vez, en todas y cada una de las entradas del array. También difieren en el formato:

```
# Lista:
[a1, a2, ..., an]

# Array
[a1 a2 ... an]

```

Una lista x (o cualquier objeto iterable como una tupla) se puede convertit en un array con entradas de tipo str,float, int o bool, de la siguiente manera:

```
x_array = np.array(x, dtype = tipo)

```

La mayoria de lo aplicable a las listas se aplica a los arryas, salvo el método append.

12.3.1. Funciones de numpy

Algunas de las funciones de numpy son:

- array(): convierte el arg en un array
- min(array=z, axis=j, out=x): obtiene el valor mínimo de los valores en la dimensión j del array z y guarda la salida, con dimensión n-1 en x (si no se especifican los argumentos tomará el mínimo de todos los elementos del array)
- max(array=z, axis=j, out=x): hace lo mismo que min() pero tomando el máximo.
- argmin(array=z, axis=j, out=x): hace lo mismo que min() pero en lugar de entregar los valores mínimos, entrega sus índices de posición.
- argmax(array=z, axis=j, out = x): hace lo mismo que argmin() pero con los valores máximos.
- mean(array=z, axis=j, out=x): hace lo mismo que las funciones anteriores pero entregando el promedio.
- std(array=z, axis=j, out=x, ddof=): hace lo mismo que las funciones anteriores pero entregando la desviación estándar (desvest), el argumento ddof sirve para establecer si se calcula la desvest muestral o poblacional (ddof=0 corresponde a la poblacional y ddof=1 corresponde a la muestral que es la que se usa en física).
- sum(array=z, axis=j, out=x, initial=n): hace lo mismo que las funciones anteriores pero entregando la suma de los primeros n valores.
- round(array=z, decimal=, out=x): redondea al número de decimales especificado en su argumento.
- clip(array=z, min=, max=, out=x): pone límites inferiores y superiores a las entradas del array. Si las entradas superan estos límites son sustituidos por min y max respectivamente.
- copy(): crea una copia del array al que se coloque como arg.

12.3.2. Métodos y rutinas de numpy

Un par de métodos útiles son:

astype(): al aplicarse a un array cambia la clase de sus elementos a la clase especificada en el argumento. Por ejemplo cambia las entradas de enteros a strings.

tolist(): convierte el array al que se le aplique en una lista.

Los arrays tienen los siguientes atributos o propiedades:

shape: proporciona la forma del array "x". Por ejemplo, si se tiene una matriz de nxm entonces print(x.shape) arrojará (n,m).

size: proporciona la cantidad total de elementos del array.

Para generar un array se tiene las siguientes rutinas:

- .empty(shape=(m1,m2,...,mn), dtype=float): crea un arreglo "vacío" de m1×m2×mn (puede contener datos basura de la memoria RAM).
- .zeros(shape=, dtype=float): crea un arreglo con puros ceros.
- .ones(shape, dtype=float): crea un arreglo con puros unos.
- .full(shape=, valor=a, dtype=) : crea un arreglo con puras a.
- .identity(n, dtype=float): crea un una matriz identidad de nxn.
- .arange(inicio=, final=, paso=, dtype=float): funciona como la función range pero genera un array en vez de una lista.
- .linspace(inicio=,final=, n): genera una partición del intervalo [inicio,final] en n partes iguales (Por default n=50):

Dado un array x, este puede manipularse con las siguientes rutinas:

- .reshape(x, shape=): convierte al array x en un array con las dimensiónes indicadas en shape.
- .roll(x, shift, axis=): recorre todos los elementos del array en el sentido dado por shift (1, hacia la derecha (abajo) y -1 hacia la izquierda (arriba)) y en la dirección dada por axis.

- .split(x, n, axis=): parte al array x a lo largo de la dirección indicada por axis, en n partes iguales.

- .concatenate((x1...xn), axis=): une los arrays x1, ... , xn a lo largo de las direcciones dadas por axis, siempre y cuando en dicha dirección se tenga la misma cantidad de elementos.

Si el array x de m1×m2×$\cdots$×mn (Es de dimensión n), para elegir las entradas de x se usa x[i1,i2,...,in]. El valor de axis=0 hace referencia a la dirección horizontal (i1 en x) y axis=1 a la vertical (i2 en x). En general, axis=k-1 hace referencia a la dirección ik en x.

12.3.3. Otras funciones de numpy

diff(): En el caso de un array de la forma x=(a1 ... an), esta función entrega un array de la forma y=(b1 ... bn-1) con bi=ai+1-ai. De este modo se puede calcular la derivada de alguna función f como:

```
import numpy as np

# Dominio, imagen e incremento
x = linspace(inicio, fin, pasos)
y = f(x)
h = (fin-inicio)/pasos

# Calculo de la derivada
dy = np.diff(y)/h

```

trapz(y, x): integra un conjunto de puntos de una función y=f(x) mediante el método del trapecio.

interp(I,x,y): dado un conjunto de puntos de una función y=f(x), interpola linealmente los valores de la función en un conjunto de puntos I dentro del dominio.

12.3.4. Numeros aleatorios con numpy

Para generar arreglos de números aleatorios se tienen las siguientes funciones:

- random.rand(a1, a2, ..., an): genera un arreglo de a1 x a2 ...x an de números aleatorios entre 0 y 1.

- random.randint(l,h,size=, dtype=): genera un arreglo de dimensión establecida por size, de valores aleatorios entre l y h-1 definidos por dtype.

- random.normal(loc=, scale=, size=n): genera una muestra de n números aleatorios provenientes de una distribución normal con un promedio y desviación estándar definidos por loc y scale respectivamente. También hay otras distribuciones como poisson, binomial, etc.

- random.choice(x): Elige de manera "aleatoria" un elemento de la lista (o tupla) x.

Una forma más reciente de generar los números aleatorios es primero estableciendo una semilla. Esto permite reproducir los resultados a pesar de que se este trabajando con "números aleatorios". Esto se hace con:

```
semilla = np.random.default_rng()

```

donde en el arg se coloca la smilla (seed) deseada, de no colocarla se toma una al azar. Con la semilla establecida, los arreglos se generan con:

```
semilla.random((a1, ..., an))

```

donde se puede cambiar random por integer, normal, etc. con los argumentos correspondientes. Para obtener números aleatorios distintos cada vez, se tiene que cambiar elegir una semilla aleatoriamente en cada llamada. Esto se hace con:

```
semilla = np.random.default_rng(random.seed())

```

12.3.5. Polinomios con numpy

Numpy tiene definida la clase de polinomios. Estos se generan con:

```
polinomio = np.poly1d((an, ..., a0))

```

donde (an, ..., a0) contiene los coeficientes del polinomio de la mayor potencia a la menor.

Los polinomios tienen los siguientes métodos:

- .deriv(m=): entrega la derivada de orden m del polinomio.

- .integ(m=, k=): entrega la integral de orden m del polinomio, considerando una constante de integración k.

- .polyfit(x,y,n,w,cov=): realiza un ajuste polinomial de grado n a los datos (x,y) con los pesos w y (si cov=True) entrega la matriz de covarianza del ajuste.

12.4. re (regular expressions)

Módulo que sirve para encontrar y manejar expresiones regulares en cadenas de texto.

12.5. Pandas

Módulo para análisis de datos. Es similar a numpy en el sentido de que se manejan "arreglos de datos" en este caso data frames, df, pero con un enfoque menos matemático y más de manipulación de una gran cantidad de datos (como si fueran hojas de Excel).
Para generar un df a partir de un archivo csv (comma separated values) se usa:

```
import pandas as pd

df = pd.read_csv(ruta\\archivo.csv)

```

se usa excel en lugar de csv para abrir una hoja de excel.

En el caso de que el archivo no cuente con una línea que defina los nombres de las columnas (la primera fila) se agrega el argumento names = ['nom1', ... ,'nomn'], el cual es una lista con los nombres de las columnas.

Para generar un data frame a partir de un diccionario se usa

```
df = pd.DataFrame(diccionario)
```

Para hacer referencia a una columna del df, se usa df['nombre_columna']

Para agregar una lista de datos x a un df como una columna con cierto nombre, se usa:

```
df['nombre']=x
```

12.6. Métodos de un data frame

- .head(i): muestra las primeras i filas del df. Si no se coloca i se muestran 5 filas por defecto.
- .tail(i): muestra las últimas i filas del df. Si no se coloca i se muestran 5 filas por defecto.
- .sort_values(by='nombre_columna', ascending=): Ordena las filas según los valores de la columa deseada y en orden ascendente si ascending=True y en descendente si ascending=False.
- .sort_index(ascending=): Ordena las filas según los valores de los índices, en orden ascendente si ascending=True y en descendente si ascending=False.
- .concat([df1, ..., dfn]): concatena los df df1, ..., dfn. Estos tienen que contar con las mismas columnas.
- .shape: entrega una tupla de la forma (filas, columnas).
- .describe(): proporciona la información y una estadística rápida del data frame.
- .loc[i,'nombre_columna']: Selecciona el valor de la fila i, de la columna 'nombre_columna'.
- .iloc[i,j]: Selecciona el valor de la fila i, de la columna i.
- .groupby('columna'): agrupa los datos por los valores de columna. Se tiene que aplicar seguido del método .describe() para obtener las estadísticas de cada grupo.

  En particular se puede usar:

  ```
  df.groupby('col1')['col2']
  ```

  esto proporciona todos los valores de 'col2' que tienen el mismo valor de 'col1'. Se tiene que aplicar seguido de alguno de los métodos .mean(), .sum(), . count(). para obtener valores de cada grupo.
- .pivot_table(index=['col1', ..., 'coln'], columns='colum', aggfunction='funcion'): agrupa los datos y genera una tabla con los valores. Conviene revisar el ejemplo de la documentación para ver lo que hace.
- .to_excel(): convierte el df en una hoja de excel. Se usa csv en lugar de excel para generar un archivo csv.
- .dropna(): se eliminan las filas que tienen valores NAN.

12.7. Módulos propios

Es cualquier script de Python creado por uno mismo, externo al script en el que se está trabajando actualmente. De preferencia debe contener solo funciones, métodos, clases, etc. más no instrucciones que se ejecuten sin ser llamadas (por ejemplo un print). Para hacer uso de los objetos de un módulo propio, este debe encontrarse en la misma carpeta que el script con el que se trabaja actualmente. Para importarlo también se usa **import, from, as** etc. El módulo también puede encontrarse dentro de una carpeta con la misma ubicación del script en donde quiere importarse, en este caso, debe usarse:

```
import nombre_carpeta.nombre modulo as ....
```

Si el módulo se encuentra una carpeta más atras de donde script en donde quiere importarse o en alguna otra ruta, esta debe agregarse en **path** con ayuda de la función **sys()** y el método **.append()**. Esto se detalla en [2]. Que mejor que guardar los módulos en la carpeta donde se está trabajando.

13. Paquetes

Es una carpeta que contiene varios módulos. Para crearlo se tiene que generar una carpeta con un archivo .py vacío y con nombre __init__. Para acceder a las funciones de los módulos se hace igual que en el caso del módulo dentro de una carpeta, mencionado en la sección 12.7.

Referencias

[1] E. L. Pineda, "Python desde cero." https://www.youtube.com/@PythonDesde0/videos, octubre 2021. Accedido en junio de 2024.

[2] S. Dalto, "Curso de python desde cero (completo)." https://docs.python.org/es/3/tutorial/modules.html#id3, enero 2023. Accedido en junio de 2024.

[3] Python™, "Módulos." https://docs.python.org/es/3/tutorial/modules.html#id3, junio 2024. Accedido en junio de 2024.