[Pickling in Python - The Very Basics](https://ianlondon.github.io/blog/pickling-basics/" \o "Pickling in Python - The Very Basics)

<https://ianlondon.github.io/blog/pickling-basics/>

You just ran through a time-consuming process to load a bunch of data into a python object. Maybe you scraped data from thousands of websites. Maybe you computed a zillion digits of pi. If your laptop battery dies or if python crashes, your information will be lost.

Pickling allows you to save a python object as a binary file on your hard drive. After you pickle your object, you can kill your python session, reboot your computer if you want, and later load your object into python again.

You could back up your pickle file to Google Drive or DropBox or a plain old USB stick if you wanted. You could email it to a friend.

A word of warning: don’t load pickles that you don’t trust. Malicious people can make malicious pickles that may execute unexpected code on your computer (SQL injection, password brute forcing, etc). Stay away from bad pickles.

import pickle

# make an example object to pickle

some\_obj = {'x':[4,2,1.5,1], 'y':[32,[101],17], 'foo':True, 'spam':False}

To save a pickle, use pickle.dump.

A convention is to name pickle files \*.pickle, but you can name it whatever you want.

Make sure to open the file in 'wb' mode (write binary). This is more cross-platform friendly than 'w' mode (write text) which might not work on Windows, etc.

with open('mypickle.pickle', 'wb') as f:

pickle.dump(some\_obj, f)

# note that this will overwrite any existing file

# in the current working directory called 'mypickle.pickle'

For the purposes of demonstration, I’ll delete the original object from memory to show you that it’s really gone.

del some\_obj

print some\_obj

---------------------------------------------------------------------------

NameError Traceback (most recent call last)

<ipython-input-3-3080f97d7e85> in <module>()

1 del some\_obj

2

----> 3 print some\_obj

NameError: name 'some\_obj' is not defined

Loading the pickled file from your hard drive is as simple as pickle.load and specifying the file path:

with open('mypickle.pickle') as f:

loaded\_obj = pickle.load(f)

print 'loaded\_obj is', loaded\_obj

loaded\_obj is {'y': [32, [101], 17], 'x': [4, 2, 1.5, 1], 'foo': True, 'spam': False}

<https://ianlondon.github.io/blog/pickling-basics/>

**ITERTOOLS**

### Bucles eficientes con Itertools

https://python-para-impacientes.blogspot.com/2015/08/bucles-eficientes-con-itertools.html

**Itertools** es un módulo de la librería estándar de Python que incorpora funciones que devuelven objetos iterables, es decir, estructuras de datos basadas en elementos que pueden recorrerse secuencialmente y que pueden utilizarse en procesos repetitivos (bucles).   
  
Estas funciones fueron diseñadas para una ejecución rápida, haciendo un uso eficiente de la memoria, con la idea de resolver algoritmos basados en bucles más complicados que aquellos que habitualmente se suelen implementar en un programa para recorrer los elementos de una lista, diccionario, etc.

### Funciones que devuelven iterables infinitos

Agrupa un conjunto de funciones **Itertools** que devuelven un iterable que no se interrumpirá si no se fuerza un final, por ejemplo, cuando se cumpla una determinada condición.

### count()

Devuelve un objeto iterable en la que el primer elemento tendrá el valor inicial (***start***) y los sucesivos se irán incrementando/decrementado con el valor del paso (***step***), de manera ininterrumpida.

***itertools.count(start=0 [, step=1])***

En el siguiente ejemplo cuando se alcanza un valor determinado se fuerza la interrupción del bucle:

from itertools import \*

for valor in count(5, 3):

print(valor, end = ' ')

if valor == 20: break

Salida: 5 8 11 14 17 20

### cycle()

Devuelve un objeto iterable con los elementos (de principio a fin) del iterable de entrada, que se reproducirán una y otra vez mientras no se fuerce un final.

***itertools.cycle(iterable)***

A continuación, varios ejemplos:   
  
**cycle() con una cadena:**

contador = 0

for elemento in cycle("Python"):

print(elemento, end = ' ')

contador += 1

if contador == 12: break

Salida: P y t h o n P y t h o n   
  
**cycle() con una lista:**

contador = 0

for elemento in cycle([10, 12, 14]):

print(elemento, end = ' ')

contador += 1

if contador == 5: break

Salida: 10 12 14 10 12   
  
**cycle() con un diccionario:**

contador = 0

for elemento in cycle({'x':1, 'y':2, 'z': 3}):

print(elemento, end = ' ')

contador += 1

if contador == 9: break

Salida: x z y x z y x z y

### repeat()

Devuelve el objeto completo, una y otra vez, de manera indefinida a menos que se especifique el número de veces (***times***) que hay que ejecutar el bucle.

***itertools.repeat(object[, times])***

A continuación, varios ejemplos:   
  
**repeat() con un entero:**

for elemento in repeat(3, 5):

print(elemento, end = ' ')

Salida: 3 3 3 3 3   
  
**repeat() con map():**

print(list(map(pow, range(5), repeat(3))))

Salida: [0, 1, 8, 27, 64]

### Funciones que devuelven iterables que finalizan

Agrupa funciones del módulo **Itertools** que devuelven iterables que finalizan con la secuencia de entrada más corta.

### accumulate()

La función devuelve un iterable con sumas acumuladas o totales acumulados derivados de los resultados obtenidos al aplicar alguna función (***func***).  
  
Los elementos del objeto iterable se evalúan tomando el primer elemento con el segundo; después el resultado obtenido de ambos elementos con el tercero y así sucesivamente.

***itertools.accumulate(iterable[, func])***

La función que se incluya debe tener dos argumentos (uno por cada elemento que se evalúa) y los elementos de la entrada iterable, necesariamente, serán del tipo que acepte dicha función. Además, si la entrada iterable está vacía la salida también lo estará.   
  
A continuación, varios ejemplos:   
  
**accumulate() con la función implícita que acumula sumas:**

for acumulado in accumulate([1, 2, 3, 4, 5]):

print(acumulado, end = ' ')

Salida: 1 3 6 10 15   
  
**accumulate() con función max para 'acumular' el valor máximo:**

for valor\_maximo in accumulate([1, 3, 2, 5, 4], max):

print(valor\_maximo, end = ' ')

Salida: 1 3 3 5 5   
  
**acumulate() con función lambda:**

for diferencia in accumulate([10, 30, 50], lambda a, b: b-a):

print(diferencia, end = ' ')

Salida: 10 20 30

### chain()

La función devuelve un iterable construido con todos los elementos de los objetos iterables de entrada.

***itertools.chain(\*iterables)***

for elemento in chain([1, 2], [3, 4, 5]):

print(elemento \*\* 2, end = ' ')

Salida: 1 4 9 16 25

### chain.from\_iterable()

Es similar a **chain()** pero solo admite un argumento iterable.

***classmethod chain.from\_iterable(iterable)***

for elemento in chain.from\_iterable(["un","dos","tres"]):

print(elemento, end = ' ')

Salida: u n d o s t r e s

### compress()

La función devuelve un iterable a partir de los elementos de **data** que tienen cada uno un valor asociado en **selectors** que actúan como un filtro. Todos los elementos con el valor ***True*** o ***1***serán los utilizados para construir el iterable. El proceso finaliza cuando se agoten los elementos en **data** o en **selectors**.

***itertools.compress(data, selectors)***

for elemento in compress("iterable", [1,0,1,0,1,0,0,1]):

print(elemento, end = ' ')

Salida: i e a e

### dropwhile()

Construye un iterable a partir del iterable de entrada sin devolver ningún elemento hasta que la condición del **predicado** sea falsa. Después de ese momento se devuelven todos los elementos que resten.

***dropwhile(predicate, iterable)***

for elemento in dropwhile(lambda valor: valor == 'x',

['x','x','y','z','x','x']):

print(elemento, end = ' ')

Salida: y z x x

### filterfalse()

Esta función devuelve un iterable con los elementos del iterable de entrada que no cumplan la condición expresada en el **predicado**. Si el predicado es **None** devuelve los elementos con valor **False** o **0**.

***itertools.filterfalse(predicate, iterable)***

for elemento in filterfalse(lambda valor: valor == 'x',

['x','x','y','z','x','x']):

print(elemento, end = ' ')

Salida: y z

### groupby()

Devuelve un iterable con los elementos del iterable de entrada agrupados por el dato utilizado como clave (**key**).

***itertools.groupby(iterable, key=None)***

A continuación, varios ejemplos:   
  
**groupby() con lista ordenada:**  
  
En el siguiente ejemplo la lista de tuplas contiene ciudades de diferentes países ordenados alfabéticamente:

ciudades = [("Bolivia", "Sucre"), ("Bolivia", "La Paz"),

("Chile", "Valdivia"), ("Chile", "Arica"),

("España", "Cádiz"), ("Perú", "Cusco"),

("Perú", "Lima")]

for clave, grupo in groupby(ciudades, lambda x: x[0]):

print(clave, list(grupo))

Salida:  
Bolivia [('Bolivia', 'Sucre'), ('Bolivia', 'La Paz')]  
Chile [('Chile', 'Valdivia'), ('Chile', 'Arica')]  
España [('España', 'Cádiz')]  
Perú [('Perú', 'Cusco'), ('Perú', 'Lima')]

**groupby() y itemgetter() con lista desordenada:**  
  
En el siguiente ejemplo la lista de tuplas contiene ciudades de diferentes países que no están ordenados alfabéticamente.   
  
Para que se pueda agrupar la información de cada país es necesario ordenar la lista. Para ello se utiliza la función **sorted()** con la función **itemgetter()** del módulo **operator**. Esta función permite establecer el criterio de ordenación. En el ejemplo, como cada tupla contiene dos elementos (país, ciudad) **itemgetter(0)** establece que el criterio de orden será el primer elemento de la tupla, es decir, el país.

from operator import itemgetter

ciudades = [("Perú", "Cusco"), ("Chile", "Valdivia"),

("Bolivia", "Sucre"), ("Bolivia", "La Paz"),

("España", "Cádiz"), ("Chile", "Arica"),

("Perú", "Lima")]

ciudades = sorted(ciudades, key=itemgetter(0))

for clave, grupo in groupby(ciudades, itemgetter(0)):

print(clave, list(grupo))

Salida:  
Bolivia [('Bolivia', 'Sucre'), ('Bolivia', 'La Paz')]  
Chile [('Chile', 'Valdivia'), ('Chile', 'Arica')]  
España [('España', 'Cádiz')]  
Perú [('Perú', 'Cusco'), ('Perú', 'Lima')]

### islice()

La función devuelve un iterable con una selección de elementos del iterable de entrada. Permite retornar un número de elementos partiendo desde el inicio del iterable o un rango específico.

***itertools.islice(iterable, stop)***

A continuación, algunos ejemplos:   
  
**Devuelve elementos partiendo desde el comienzo:**

for elemento in islice("KLMNOPQRST", 5):

print(elemento, end = ' ')

Salida: K L M N O

***itertools.islice(iterable, start, stop [, step])***

**Devuelve los elementos que hay desde una posición inicial a una final:**

for elemento in islice("KLMNOPQRST", 5, 7):

print(elemento, end = ' ')

Salida: P Q

**Devuelve los elementos que hay desde una posición inicial a una final, separados entre sí por un valor fijo de elementos:**

for elemento in islice("1234567890", 0, 8, 2):

print(elemento, end = ' ')

Salida: 1 3 5 7

### starmap()

Construye un objeto iterable aplicando una función que utiliza como argumentos los elementos del iterable de entrada, devolviendo una secuencia con los resultados obtenidos.

***itertools.starmap(function, iterable)***

En el ejemplo se recorre una lista de tuplas y se construye el iterable a devolver con el valor más alto de los elementos que hay en cada una de ellas:

for elemento in starmap(max,

[(10,2),(2,32),(63,54),(4,45)]):

print(elemento, end = ' ')

Salida: 10 32 63 45

### takewhile()

Construye un iterable a partir del iterable de entrada devolviendo elementos mientras la condición del predicado es verdadera. En el momento que cambie a falsa no devolverá más elementos aunque exista alguno que cumpla la condición.

***itertools.takewhile(predicate, iterable)***

En el ejemplo se recorre una lista de cadenas comprobando si su longitud es igual a 1:

for elemento in takewhile(lambda x: len(x) == 1,

['a','b','ab','bc','c']):

print(elemento, end = ' ')

Salida: a b

### tee()

Devuelve varios iterables independientes (por defecto, 2) sobre la base de una sola entrada original.

***itertools.tee(iterable, n=2)***

a, b = tee([1,2,3,4])

for elemento1, elemento2 in zip(a, b):

print("a:", elemento1)

print("b:", elemento2)

Salida:  
a: 1  
b: 1  
a: 2  
b: 2  
a: 3  
b: 3

### zip\_longest()

Devuelve un iterable que agrega elementos de cada uno de los iterables de entrada. Si los iterables de entrada tienen distinta longitud se completará utilizando el valor de **fillvalue**, hasta que se alcance el final del iterable que tenga una longitud mayor.

***itertools.zip\_longest(\*iterables, fillvalue=None)***

for elemento in zip\_longest(['x','y','z'], ['0','1'],

fillvalue='#'):

print(elemento, end = ' ')

Salida: ('x', '0') ('y', '1') ('z', '#')

### Generadores para combinatoria

Agrupa una serie de funciones que generan iterables como resultado de operaciones de combinatoria en las que se utilizan otros iterables de entrada.

### product()

Realiza el producto cartesiano con los elementos de los iterables de entrada devolviendo un iterable basado en tuplas con el resultado.

***itertools.product(\*iterables, repeat=1)***

for elemento in product("XYZ", "mn"):

print(elemento, end = ' ')

Salida: ('X', 'm') ('X', 'n') ('Y', 'm') ('Y', 'n') ('Z', 'm') ('Z', 'n')

for elemento in product("X", repeat = 4):

print(elemento, end = ' ')

Salida: ('X', 'X', 'X', 'X')

for elemento in product("XYZ", repeat = 2):

print(elemento, end = ' ')

Salida: ('X', 'X') ('X', 'Y') ('X', 'Z') ('Y', 'X') ('Y', 'Y') ('Y', 'Z') ('Z', 'X') ('Z', 'Y') ('Z', 'Z')

### permutations()

Devuelve un objeto iterable basado en tuplas con permutaciones de longitud **r** a partir de los elementos del objeto de entrada.

***itertools.permutations(iterable, r=None)***

for elemento in permutations("123", 2):

print(elemento, end = ' ')

Salida: ('1', '2') ('1', '3') ('2', '1') ('2', '3') ('3', '1') ('3', '2')

### combinations()

Devuelve un objeto iterable basado en tuplas con las combinaciones sin repetición posibles, de una longitud **r**, a partir de los elementos del objeto de entrada.

***itertools.combinations(iterable, r)***

for elemento in combinations("123", 2):

print(elemento, end = ' ')

Salida: ('1', '2') ('1', '3') ('2', '3')

### combinations\_with\_replacement()

Devuelve un objeto iterable basado en tuplas con las combinaciones con repetición posibles, de una longitud **r**, a partir de los elementos del objeto de entrada.

***itertools.combinations\_with\_replacement(iterable, r)***

for elemento in combinations\_with\_replacement("123", 2):

print(elemento, end = ' ')

Salida: ('1', '1') ('1', '2') ('1', '3') ('2', '2') ('2', '3') ('3', '3')

PANDAS

https://jarroba.com/pandas-python-ejemplos-parte-i-introduccion/

Pandas es una librería de python destinada al análisis de datos, que proporciona unas estructuras de datos flexibles y que permiten trabajar con ellos de forma muy eficiente. Pandas ofrece las siguientes estructuras de datos:

* **Series:** Son arrays unidimensionales con indexación (arrays con índice o etiquetados), similar a los diccionarios. Pueden generarse a partir de diccionarios o de listas.
* **DataFrame:** Son estructuras de datos similares a las tablas de bases de datos relacionales como SQL.
* **Panel, Panel4D y PanelND:** Estas estructuras de datos permiten trabajar con más de dos dimensiones. Dado que es algo complejo y poco utilizado trabajar con arrays de más de dos dimensiones no trataremos los paneles en estos tutoriales de introdución a Pandas.

Para ver los ejemplos que vamos a mostrar en este tutorial y en el resto relacionados con la librería de Pandas, es necesario descargarnos esta librería (y la librería de Numpy). Para ello lo podemos descargar a través del repositorio de paquetes PyPi (con pip) de la siguiente manera:

$ pip install pandas

$ pip install numpy

Vamos a ver a continuación ejemplos básicos con Series y DataFrame:

Lo primero que debemos hacer es importar la libreria de Pandas:

import pandas as pd

Por convenio (de la comunidad de desarrollares) se pone "**pd**" como alias de la librería Pandas.

El primer ejemplo que vamos a poner va a ser el de definir una estructura de datos "***Series***" que como ya comentamos es un array de datos unidimensional con idexación. Las "Series" se definen de la siguiente manera:

serie = pd.Series(data, index=index)

Es decir, que en el primer parámetro le indicamos los datos del array y en el segundo parámetro los índices. Veamos un ejemplo de como crear una estructura "Series" con los integrantes de la selección Española de fútbol que ganó el mudial del año 2010, en el que tenemos como 'data' sus nombres y como índice su dorsal:

spanishPlayers = pd.Series(

['Casillas', 'Ramos', 'Pique', 'Puyol', 'Capdevila', 'Xabi Alonso', 'Busquets', 'Xavi Hernandez', 'Pedrito',

'Iniesta', 'Villa'], index=[1, 15, 3, 5, 11, 14, 16, 8, 18, 6, 7])

print "Spanish Football Players: \n%s" % spanishPlayers

Como salida a este fragmento de código en el que se va a imprimir por pantalla la Serie es la siguiente, en la que verémos lo anteriormente mencionado:

Spanish Football Players:

1 Casillas

15 Ramos

3 Pique

5 Puyol

11 Capdevila

14 Xabi Alonso

16 Busquets

8 Xavi Hernandez

18 Pedrito

6 Iniesta

7 Villa

dtype: object

En el siguiente caso; en el que no le indiquemos los índices de forma explícita, no generará los índices de forma automática empezando desde el valor cero:

spanishPlayers = pd.Series(

['Casillas', 'Ramos', 'Pique', 'Puyol', 'Capdevila', 'Xabi Alonso', 'Busquets', 'Xavi Hernandez', 'Pedrito',

'Iniesta', 'Villa'])

print "Spanish Football Players: \n%s" % spanishPlayers

Como salida tenemos lo siguiente:

Spanish Football Players:

0 Casillas

1 Ramos

2 Pique

3 Puyol

4 Capdevila

5 Xabi Alonso

6 Busquets

7 Xavi Hernandez

8 Pedrito

9 Iniesta

10 Villa

dtype: object

También podemos crearnos una estructura de datos "Series" a partir de una lista o de un diccionario. Si la construimos a partir de una lista nos pondrá los índices por defecto y si lo creamos a partir de un diccionario, pondrá como índices las claves. Vamos a ver a continuación un ejemplo de como crear una Serie a partir de un diccionario y además vamos a ver como insertar en esta serie un nuevo elemento:

dictPlayers = {1: 'Casillas', 15: 'Ramos', 3: 'Pique', 5: 'Puyol', 11: 'Capdevila', 14: 'Xabi Alonso',

16: 'Busquets', 8: 'Xavi Hernandez', 18: 'Pedrito', 6: 'Iniesta', 7: 'Villa'}

players2series = pd.Series(dictPlayers)

*# Insert new player*

players2series[10] = 'Cesc'

print "Spanish Football Players through dictionary: \n%s" % players2series

Como salida a este fragmento de código tenemos lo siguiente:

Spanish Football Players through dictionary:

1 Casillas

3 Pique

5 Puyol

6 Iniesta

7 Villa

8 Xavi Hernandez

11 Capdevila

14 Xabi Alonso

15 Ramos

16 Busquets

18 Pedrito

10 Cesc

dtype: object

Vamos a pasar a continuación a ver un ejemplo con la estructura de datos "**DataFrame**". Como ya se ha comentado es una estructura de datos similar a una tabla de una base de datos relacionar, una tabla de excel, etc. y como tal se pueden hacer muchas operaciones como las que se harían con consultas a tablas de bases de datos o en excel (este tipo de operaciones se pueden ver en los siguiente tutoriales: ["DataFrame: Lectura y Escritura, Mergeo de DataFrame's y GroupBy](https://jarroba.com/pandas-python-ejemplos-parte-ii-io-merge-groupby/)" y "[Operaciones de "Pivot\_table" con DataFrame's](https://jarroba.com/pandas-python-ejemplos-parte-iii-pivot_table/)").

Para construir un DataFrame se puede hacer de diferentes formas, como por ejemplo a partir de una lista, de un diccionario, de una Serie, de otro DataFrame, leyendo una tabla excel, csv, etc. Vamos a ver a continuación como construiríamos un DataFrame con datos de los integrantes de la selección Española de Fútbol:

spanishPlayersDF = pd.DataFrame(

{

'name': ['Casillas', 'Ramos', 'Pique', 'Puyol', 'Capdevila', 'Xabi Alonso', 'Busquets', 'Xavi Hernandez',

'Pedrito', 'Iniesta', 'Villa'],

'demarcation': ['Goalkeeper', 'Right back', 'Centre-back', 'Centre-back', 'Left back', 'Defensive midfield',

'Defensive midfield', 'Midfielder', 'Left winger', 'Right winger', 'Centre forward'],

'team': ['Real Madrid', 'Real Madrid', 'FC Barcelona', 'FC Barcelona', 'Villareal', 'Real Madrid',

'FC Barcelona', 'FC Barcelona', 'FC Barcelona', 'FC Barcelona', 'FC Barcelona']

}, columns=['name', 'demarcation', 'team'], index=[1, 15, 3, 5, 11, 14, 16, 8, 18, 6, 7]

)

Vemos como en primer lugar le pasamos como parámetro un diccionario que contiene como claves los nombres de las columnas y como valores una lista con los datos que tendrá cada columna de la tupla. En segundo lugar le pasamos el nombre de las columnas que coinciden con las claves del diccionario con los datos y por último el índice de cada una de las tuplas. Si imprimimos el contenido de este DataFrame tenemos los siguiente:

name demarcation team

1 Casillas Goalkeeper Real Madrid

15 Ramos Right back Real Madrid

3 Pique Centre-back FC Barcelona

5 Puyol Centre-back FC Barcelona

11 Capdevila Left back Villareal

14 Xabi Alonso Defensive midfield Real Madrid

16 Busquets Defensive midfield FC Barcelona

8 Xavi Hernandez Midfielder FC Barcelona

18 Pedrito Left winger FC Barcelona

6 Iniesta Right winger FC Barcelona

7 Villa Centre forward FC Barcelona

Como resultado tenemos una estructura de dato similar a la de una tabla de una base de datos relacional o de un documento excel o csv. Con esta estructura de datos se pueden hacer muchas operaciones como las que haríamos en una base de datos o en un documento excel.

Por último vamos a ver como insertar un nuevo elemento en este DataFrame, que lo haríamos de la siguiente manera con el método "loc()":

*# Insert new player*

spanishPlayersDF.loc[10] = ['Cesc', 'Forward', 'Arsenal']

Vemos que le indicamos el índice que tiene que tener en la tabla (como lo haríamos con un diccionario) y luego con una lista indicamos los valores de las columnas. Como resultado tendríamos un nuevo elemento de el DataFrame:

Spanish Football Players DataFrame:

name demarcation team

1 Casillas Goalkeeper Real Madrid

15 Ramos Right back Real Madrid

3 Pique Centre-back FC Barcelona

5 Puyol Centre-back FC Barcelona

11 Capdevila Left back Villareal

14 Xabi Alonso Defensive midfield Real Madrid

16 Busquets Defensive midfield FC Barcelona

8 Xavi Hernandez Midfielder FC Barcelona

18 Pedrito Left winger FC Barcelona

6 Iniesta Right winger FC Barcelona

7 Villa Centre forward FC Barcelona

10 Cesc Forward Arsenal

### **CONCLUSIONES:**

La finalidad de este tutorial es dar a conocer las dos estructuras de datos (Series y DataFrame's) más utilizadas de la librería Pandas y como tál se han realizado ejemplos muy básicos. Se pueden hacer muchisimas cosas más; algunas de las cuales las explicamos en los tutoriales ["DataFrame: Lectura y Escritura, Mergeo de DataFrame's y GroupBy](https://jarroba.com/pandas-python-ejemplos-parte-ii-io-merge-groupby/)" y "[Operaciones de "Pivot\_table" con DataFrame's](https://jarroba.com/pandas-python-ejemplos-parte-iii-pivot_table/)". Si queréis saber más sobre la librería de Pandas lo podéis hacer mirando en el siguiente enlace su extensa y detallada documentación: <http://pandas.pydata.org/pandas-docs/stable/>