# Lecciones aprendidas programando en Python

Hecho por Gonzalo Fernandez y Claudio de EFX

* **Importar otros archivos .py**

Es más prolijo importar aquellas funciones que necesitamos antes que importar toda la librería

Bien

Import nombre\_archivo\_ejemplo as nombre

nombre.funcion

from nombre\_archivo\_ejemplo import funcion

funcion

Mal

from nombre\_archivo\_ejemplo import \*

* **Para verificar si una lista tiene aunque sea un valor/posición**

Bien

If lista:

Mal 1

If len(lista) > 0:

Mal 2

If lista != []:

En caso de necesitar saber si la lista tiene más de dos valores/posiciones, usar función len(), sino alcanza con preguntar por la lista en la condición.

* **Para verificar por valores True o False**

Bien

If x:

Mal

If x == True:

Los nombres True y False son instancias de tipo de datos bool (valores booleanos). Al igual que None, solo hay una instancia de cada uno.

| **False** | **True** |
| --- | --- |
| False (== 0) | True (== 1) |
| "" (empty string) | any string but "" ("anything") |
| 0, 0.0 | any number but 0 (1, 0.1, -1, 3.14) |
| [], (), {}, set() | any non-empty container ([0], (None,), ['']) |
| None | almost any object that's not explicitly False |

Tener en cuenta la tabla anterior, observar el siguiente ejemplo:

a = 0.0

b = 0

c = 1

|  |  |  |
| --- | --- | --- |
| if a:  print(‘entre’)  else:  print(‘no entre’) | if b:  print(‘entre’)  else:  print(‘no entre’) | if c:  print(‘entre’)  else:  print(‘no entre’) |
| Resultado = ‘no entre’ | Resultado = ‘no entre’ | Resultado = ‘entre’ |

Lo mismo ocurre con listas, diccionarios, tuplas, etc.

* **Si queremos concatenar todos los elementos de una lista de strings**

Bien

Concatenada = ‘’.join(lista)

Mal

concatenada = ''

for string in list:

concatenada += s

Ej:

lista = [‘1’,’2’,’3’,’0’,’-‘,’J’,’R’,’D’,’C’]

Concatenada = ‘’.join(lista)

----------

Concatenada = ‘1230-JRDC’

* **Asignación de etiquetas a objetos**

En Python, un "nombre" o "identificador" de una variable es como una etiqueta adjunta a un objeto (números, listas, diccionarios, etc).



a = 1

|  |  |
| --- | --- |
|  |  |

El objeto 1 (numero entero 1), se le puso la etiqueta “a”

Si reasignamos el objeto al cual apunta “a”, simplemente se mueve la etiqueta al nuevo objeto

a = 2

|  |  |  |
| --- | --- | --- |
| a2tag.png |  | 1.png |

El objeto entero original (numero entero 1) ya no tiene una etiqueta "a". Puede seguir vivo, pero no podemos acceder a él a través del nombre "a". (Cuando un objeto no tiene más referencias o etiquetas, se elimina de la memoria).

Si le asignamos al mismo objeto otra etiqueta

b = a

|  |  |
| --- | --- |
| ab2tag.png |  |

Entonces el nombre de variable “b” es una segunda etiqueta al objeto del cual es etiqueta “a” (el numero entero 2).

* **Listas por comprensión**

Para iterar y trabajar con todos los elementos de una lista, se pueden usar listas por comprensión.  
El resultado de aplicar una lista por comprensión es otra lista, la cual tiene el resultado de aplicarle una función los elementos que cumplen con la condición explicitada.

Sintaxis:

Lista\_comprension = [resultado\_a\_devolver ciclo\_for condición\_if]

Bien

Auxiliar = 0

Lista = [1,2,3]

Auxiliar = sum( [numero for numero in lista if numero > 1])

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

Auxiliar = 5

Auxiliar = 0

Lista = [1,2,3]

Auxiliar = [numero for numero in lista if numero > 1]

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

Auxiliar = [2,3]

Mal

Auxiliar = 0

Lista = [1,2,3]

For numero in lista:

If numero > 1:

Auxiliar += numero

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

Auxiliar = 5

La desventaja es que igualmente estamos usando un ciclo for para recorrer cada elemento de la lista, si bien es más performante usar listas por comprensión (ya que Python las trabaja de forma más óptima que con ciclos for y la función in()), no evitamos tener que usar un ciclo.

* **Funciones lambda usando map**

Para resolver ciertos tipos de cálculos, lo que hicimos fue crear diferentes listas que contuvieran los valores necesarios (previamente filtrados) para obtener el resultado esperado.

En un principio dichas listas se recorrían y se obtenían resultados utilizando listas por comprensión y funciones del tipo sum, len, etc.

Como fue descrito más arriba, las listas por comprensión si o si necesitan usar ciclos for.  
  
Otra forma, más óptima, es utilizar funciones lamba dentro de una función map.

Función map:

La función map recorre y aplica una función determinada sobre un conjunto de valores (Lista) y retorna una lista. No se le puede pasar por parámetro ninguna variable a la función que invocamos desde la función map. Si queremos realizar esto, se puede crear un diccionario en el scope global y luego, dentro de la función que llamamos desde la función map, hacer uso de este diccionario según corresponda.

Sintaxis:

Map(función\_a\_aplicar,lista)

Ej:

map(c\_class,row.CONSULTAS)

Supongamos que tenemos un diccionario llamado variables\_tarjetas que guarda todas las estructuras necesarias para calcular ciertas variables de tarjetas de crédito.   
Dentro de este diccionario tenemos una lista que guarda los saldos en cuotas mensuales para todas las tarjetas, filtradas por un periodo de mes en particular.

Si dijéramos que tenemos que contar la cantidad de veces que el saldo en cuotas en igual a 0:

Listas por comprensión:

Saldo\_cuotas\_eq\_0 = len([x for x in variables\_tarjetas['saldo\_cuotas\_ult'] if x == 0 ])

Map y lambda:

Saldo\_cuotas\_eq\_0 = sum(map(lambda x: 1 if x == 0 else 0,variables\_tarjetas['saldo\_cuotas\_ult']))

Si bien ambos códigos devolverán lo mismo, la diferencia reside en que no estamos usando un ciclo for.

En una línea que hagamos esto no vamos a notar mucha mejora de tiempos.

Ahora si tenemos en uso alrededor de más de 100 listas por comprensión, reemplazándolas por map y lambda, tendremos una reducción de tiempos.

Estructura de una función lambda:

Las funciones lambda pueden ser asignadas a una etiqueta o pueden usarse dentro de funciones map como vimos más arriba.

La lista de argumentos no está entre paréntesis y falta la palabra reservada return (está implícita, ya que la función entera debe ser una única expresión).

La función no tiene nombre, pero puede ser llamada mediante la etiqueta a que se le ha asignado.

Sintaxis:

Lambda parámetros(1..n): comportamiento\_funcion

Si a esto lo combinamos con las sentencias if por comprensión, se logra el resultado visto más arriba:

Sintaxis:

valor\_si\_verdadero if condición else valor\_si\_falso

Ejemplo if por compresión:

Común:

x = int()

if 8 > 7:

x = 1

else:

x = 0

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

var\_x = 1

Por comprensión:

var\_x = 1 if 8 > 7 else 0

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

var\_x = 1

Ejemplo funciones lambda:

Función común:

def f(x):

return x\*2

-----------

f(3) = 6

Funcion lambda:

g = lambda x: x\*2

--------

g(3) =6

Otra forma:

(lambda x: x\*2)(3) = 6

Ejemplo, funcion lambda con más de 1 parámetro y usando if por comprension:

Función común:

def f(x,y):

If x and y and y > 0 and x/float(y) > 0:

return 1

else:

return 0

Funcion lambda:

lambda x,y: 1 if x and y and y > 0 and x/float(y) > 0 else 0

Otro ejemplo (Funcion lambda con map e if por comprension):

vg\_b\_3t\_sdonf\_pmant\_lt100\_1 = sum(map(lambda x,y,z: 1 if x and y and z and (y-z)/float(x) < 1.0 else 0,variables\_tarjetas['monto\_min\_3t\_sdonf\_pminant'],variables\_tarjetas['saldo\_mensual\_3t\_sdof\_sdotant'],variables\_tarjetas['saldo\_financiado\_ult']))

* **Particularidad de la función map**

Si tengo las siguientes listas

A = [0,1,2,3,4]

B = [5,6,7,None,8]

C = [9,10,11,12]

Nótese que C tiene 4 elementos mientras que A y B tienen 5.

Si realizo lo siguiente:

D = map(lambda x,y,z: x if x and y and z else 0, A,B,C)

-------

D = [0,1,2,0,0]

Analizando sabemos:

El primer elemento es un 0. La lista A, su primer elemento es 0, al hacer un “if 0” devuelve “false”, por eso sale el 0.

Con el 1 y el 2 no hay mucho que ver, tanto el 6,7 como el 10,11 de las listas B y C, cumplen las condiciones de la función lambda.

El cuarto elemento es un 0. Ya que el cuarto elemento de la lista B es None (“if None” -> “False”)

Ahora, porque el 5to es un 0.

Esto se debe a que Python tiene que operar por el máximo tamaño de lista que se le pase por parámetro, agarrando el mismo elemento que itera de cada lista (Cuando agarra el 0 de la lista A agarra el 5 de la B y el 9 de la C). Como la lista C, no tiene un 5to elemento, sale por la condición false del if, ya que no puede asegurar que se cumpla para todos los elementos iterados.

Entonces, la función map me va a devolver una lista con el mismo tamaño de la lista más grande pasada por parámetro. Si se pasa una sola lista por parámetro, entonces esto no siempre se cumple.

* **Divisiones por tipo de datos float**

Supongamos el siguiente escenario

|  |  |
| --- | --- |
| A = 10  B = 5  print A/B  -----  2 | A = 11 B = 5  print A/B ----- 2 |

El resultado de la derecha debería haber dado 2,2.

Esto se debe a que el tipo de dato del divisor es entero.  
En Python cuando realizamos una división y queremos obtener un tipo de dato float, debemos indicarle que el divisor es del tipo float, ya sea casteando dicho numero como float o creándolo como float.

|  |  |
| --- | --- |
| A = 11  B = 5.0  print A/B  -----  2.2 | A = 11 B = 5  print A/float(B) ----- 2.2 |

Nos topamos con este error cuando advertimos que al realizar divisiones que dan numero decimales con parte entera 0, dentro de una condición if, daba siempre false.

Aca un ejemplo

A = 5

B = 4

El resultado de hacer

If A/B > 1

Es false, ya que 5/4 es 1,25 pero sin el casteo del divisor a float resulta que 5/4 es 1.

* **Operaciones con la instancia de objeto None**

No se puede realizar ninguna operación matemática con alguna etiqueta que apunte a una instancia del objeto None.

En otras palabras, no se pueden realizar las siguientes operaciones:

A = None

|  |  |  |  |
| --- | --- | --- | --- |
| A\*2 | A/2 | A+2 | A-2 |

Tampoco se puede concatenar None

|  |  |
| --- | --- |
| print ‘texto de prueba: ‘ + None | L = None  print ‘texto de prueba: ‘ + l |

Si se puede hacer

|  |  |
| --- | --- |
| print ‘texto de prueba: ‘ + str(None) | L = None  print ‘texto de prueba: ‘ + str(l) |

Lo cual imprimirá: ‘texto de prueba None’

Nosotros creamos nuestra propia función NVL para trabajar con el tipo de dato None.

#--------NVL-------------#

def NVL(var, val):

if var is None:

return val

else:

return var

* **Variables globales vs diccionarios**

Al trabajar con variables globales, notamos que no siempre estaban llegando a todas las funciones, es decir, no siempre dentro del scope de una función lográbamos operar con el valor de alguna variable definida como global (global nombre\_variable).  
En vez de eso, se usaron diccionarios, definidos dentro del scope global.  
De esta forma pudimos realizar un pasamano de variables transparente entre diferentes funciones a lo largo de todo el script en Python.

El manejo de diccionarios nos resultó bastante performante a la hora de trabajar con estructuras que acumulen otras estructuras de datos (listas, tuplas, diccionarios o simples variables integer, float, string) categorizados bajo una misma fuente de datos.

Además al parecer, Python es poco performante al momento de tener que mantener el seguimiento de las referencias a variables globales.

* **Concatenar dos diccionarios en uno solo**

Una práctica que nos resultó útil, fue la de concatenar los valores claves de dos diccionarios en otro.   
De esta forma podemos pasar un solo diccionario por parametro en vez de tener que pasar todos los diccionarios.

Ejemplo:

Digamos que tenemos 5 diccionarios, cada uno tiene diferentes valores claves

dict\_1 = {'dict\_1\_clave1':[],'dict\_1\_clave2':[],'dict\_1\_clave3':[]}

dict\_2 = {'dict\_2\_clave1':[],'dict\_2\_clave2':[],'dict\_2\_clave3':[]}

dict\_3 = {'dict\_3\_clave1':[],'dict\_3\_clave2':[],'dict\_3\_clave3':[]}

dict\_4 = {'dict\_4\_clave1':[],'dict\_4\_clave2':[],'dict\_4\_clave3':[]}

dict\_5 = {'dict\_5\_clave1':[],'dict\_5\_clave2':[],'dict\_5\_clave3':[]}

Si hacemos

dict\_fusion = {}

dict\_fusion = dict(dict\_fusion, \*\* dict\_1)

----------

dict\_fusion={'dict\_1\_clave1': [], 'dict\_1\_clave3': [], 'dict\_1\_clave2': []}

Entonces, continuando:

dict\_fusion = dict(dict\_fusion, \*\* dict\_2)

dict\_fusion = dict(dict\_fusion, \*\* dict\_3)

dict\_fusion = dict(dict\_fusion, \*\* dict\_4)

dict\_fusion = dict(dict\_fusion, \*\* dict\_5)

---------

dict\_fusion={'dict\_1\_clave1': [], 'dict\_2\_clave1': [], 'dict\_3\_clave3': [], 'dict\_3\_clave2': [], 'dict\_3\_clave1': [], 'dict\_5\_clave1': [], 'dict\_1\_clave3': [], 'dict\_5\_clave3': [], 'dict\_5\_clave2': [], 'dict\_4\_clave2': [], 'dict\_4\_clave3': [], 'dict\_2\_clave2': [], 'dict\_4\_clave1': [], 'dict\_1\_clave2': [], 'dict\_2\_clave3': []}

Cuando tuvimos que pasar los valores de varios diccionarios (uno por cada fuente de datos en el reporte crediticio) de un script a otro script separado, se concatenaron todos los diccionarios y se pasó uno solo por parametro.

* **Diferencia entre append y extend para trabajar con listas**

-append crea una nueva posición en la lista y agrega el objeto pasado por parámetro.

-extend crea una nueva posición en la lista si el elemento pasado por parametro es un objeto no iterable (integer, float, string, etc), si el elemento es un objeto iterable (lista), agrega cada elemento del objeto iterable pasado por parámetro en una nueva posición.

Ejemplos:

1) lista\_1 = [1,2,3,4,5]

lista\_3 = ['soy un string en la posicion 1',['soy una lista en la posicion 2']]

lista\_3.append(lista\_1)

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

lista\_3 = ['soy un string en la posicion 1', ['soy una lista en la posicion 2'], [1, 2, 3, 4, 5]]

2) lista\_1 = [1,2,3,4,5]

lista\_3 = ['soy un string en la posicion 1',['soy una lista en la posicion 2']]

lista\_3.extend(lista\_1)

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

lista\_3 = ['soy un string en la posicion 1', ['soy una lista en la posicion 2'], 1, 2, 3, 4, 5]

3) Si agregamos una lista con extend, seria lo mismo que hacer append

lista\_1 = [1,2,3,4,5]

lista\_3 = ['soy un string en la posicion 1',['soy una lista en la posicion 2']]

lista\_3.extend([lista\_1])

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

lista\_3 = ['soy un string en la posicion 1', ['soy una lista en la posicion 2'], [1, 2, 3, 4, 5]]

4) Agregamos un diccionario

dict\_1 = {'dict\_1\_clave1':[],'dict\_1\_clave2':[],'dict\_1\_clave3':[]}

lista\_3.extend(dict\_1)

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

lista\_3 = ['soy un string en la posicion 1', ['soy una lista en la posicion 2'], 'dict\_1\_clave1', 'dict\_1\_clave3', 'dict\_1\_clave2']

#

dict\_1 = {'dict\_1\_clave1':[],'dict\_1\_clave2':[],'dict\_1\_clave3':[]}

lista\_3.append(dict\_1)

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

lista\_3 = ['soy un string en la posicion 1', ['soy una lista en la posicion 2'], {'dict\_1\_clave1': [], 'dict\_1\_clave3': [], 'dict\_1\_clave2': []}]

5) Agregamos una tupla

tupla\_1 = (1,0)

lista\_3 = ['soy un string en la posicion 1',['soy una lista en la posicion 2']]

lista\_3.extend(tupla\_1)

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

lista\_3 = ['soy un string en la posicion 1', ['soy una lista en la posicion 2'], 1, 0]

#

tupla\_1 = (1,0)

lista\_3 = ['soy un string en la posicion 1',['soy una lista en la posicion 2']]

lista\_3.append(tupla\_1)

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

lista\_3 = ['soy un string en la posicion 1', ['soy una lista en la posicion 2'], (1, 0)]

6) Por ultimo agregamos una tupla de tuplas

tupla\_1 = ((1,0),(2,3))

lista\_3 = ['soy un string en la posicion 1',['soy una lista en la posicion 2']]

lista\_3.extend(tupla\_1)

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

lista\_3 = ['soy un string en la posicion 1', ['soy una lista en la posicion 2'], (1, 0), (2, 3)]

#

tupla\_1 = ((1,0),(2,3))

lista\_3 = ['soy un string en la posicion 1',['soy una lista en la posicion 2']]

lista\_3.append(tupla\_1)

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

lista\_3 = ['soy un string en la posicion 1', ['soy una lista en la posicion 2'], ((1, 0), (2, 3))]

* **Ordenamiento de listas**

Existe la función sorted que devuelve el objeto iterable pasado por parámetro ordenado según algún criterio.

Sintaxis:

sorted(iterable[, key][, reverse])

Donde:

iterable - sequence (string, tuple, list) or collection (set, dictionary, frozen set) or any iterator

reverse (Optional) - If true, the sorted list is reversed (or sorted in Descending order)

key (Optional) - function that serves as a key for the sort comparison

Ordenando una lista:

lista\_1 = [4,5,23,2,1]

lista\_1 = sorted(lista\_1)

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

lista\_1 = [1,2,4,5,23]

Ordenando una lista de listas:

lista\_1 = [[64,13,43],[6,3,5]]

lista\_1 = sorted(lista\_1)

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

lista\_1 = [[6, 3, 5], [64, 13, 43]]

Como se ve, solo ordena la lista y no los elementos que contienen las listas en su interior.

Ordenando una lista de tuplas, usando la clausula key:

def segundo(elem):

return elem[1]

lista\_1 = [(2, 2), (3, 4), (4, 1), (1, 3)]

lista\_1 = sorted(lista\_1, key= segundo)

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

lista\_1 = [(4, 1), (2, 2), (1, 3), (3, 4)]

Supongamos que tenemos una lista de diccionarios y queremos ordenarlos según el valor que tiene cada una de las claves de dichos diccionarios.

Para esto debemos tener en cuenta que todos los atributos que queramos ordenar se van a acomodar bajo el mismo criterio (mayor a menor o viceversa)

Si queremos ordenar de mayor a menor, por ejemplo, tener en cuenta que todos los atributos que se involucren en el ordenamiento quedaran bajo ese criterio.

Si necesitamos que algunos atributos se orden distintos, deberíamos cambiar la escala de valores para que vaya según el criterio de ordenamiento.

Entonces, podemos crear una función lambda que devuelva una tupla, que será por la cual ordene la función sorted, y aquellos casos que queramos que tengan un criterio de ordenamiento contrario al de otros atributos, debemos negarlos.

Siguiendo con el ejemplo, tenemos 3 diccionarios acumulados en una lista:

variables\_direcciones = {}

direcciones\_completas = []

variables\_direcciones\_1={'aportado\_fh':'2012-10','aportado\_fh\_num':'20180101','aportado\_cant':4,'cod\_postal':1406,'mes':360,'cpa':'DDCR12','geo\_nse':'NA','provincia':'P','aglomerado':'C','avg\_tc\_limite\_credito':1234.123,'pos':1,'cpa\_num':331,'impacto\_codigo\_num':12,'tasa\_mora':23123.12321,'tasa\_p\_mora':12312.4312}

direcciones\_completas.append(variables\_direcciones\_1)

variables\_direcciones\_2={'aportado\_fh':'2017-10','aportado\_fh\_num':'20180102','aportado\_cant':2,'cod\_postal':1407,'mes':180,'cpa':'DDCR12','geo\_nse':'NA','provincia':'P','aglomerado':'C','avg\_tc\_limite\_credito':1234.123,'pos':1,'cpa\_num':331,'impacto\_codigo\_num':12,'tasa\_mora':23123.12321,'tasa\_p\_mora':12312.4312}

direcciones\_completas.append(variables\_direcciones\_2)

variables\_direcciones\_3={'aportado\_fh':'2017-09','aportado\_fh\_num':'20171201','aportado\_cant':5,'cod\_postal':1400,'mes':60,'cpa':'DDCR12','geo\_nse':'NA','provincia':'P','aglomerado':'C','avg\_tc\_limite\_credito':1234.123,'pos':1,'cpa\_num':331,'impacto\_codigo\_num':12,'tasa\_mora':23123.12321,'tasa\_p\_mora':12312.4312}

direcciones\_completas.append(variables\_direcciones\_3)

Ahora queremos obtener el CPA, del diccionario que cumpla con:

menor valor de pos

mayor valor de aportado\_cant

mayor valor de aportado\_fh\_num

menor valor de cod\_postal

Como dijimos, si vamos a usar reverse = True (De mayor a menor), aquellos campos que necesitemos el menor tendremos que negarlos, y viceversa si no ponemos la cláusula reverse.

Tener en cuenta que, al ordenar por una tupla, la función sorted ira desde la primer posición hasta la última, es decir, si el valor de la primer posición de la tupla es igual a todas las demás, ordenara por el segundo campo y así hasta el final de los campos.  
Entonces si hay algún campo que sea más relevante que otro se debería meter al principio de la tupla.

Nótese que a los campos pos y cod\_postal, se les agrego un “-“ delante para negarlos y asi ordenar con un criterio opuesto al resto de los campos.

Lista\_ordenada = sorted(direcciones\_completas, key = lambda dir: (-dir['pos'],dir['aportado\_cant'],dir['aportado\_fh\_num'],-dir['cod\_postal']),reverse=True)

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

Lista\_ordenada = [{'aportado\_fh': '2017-09', 'impacto\_codigo\_num': 12, 'geo\_nse': 'NA', 'pos':1, 'avg\_tc\_limite\_credito': 1234.123, 'mes': 60, 'cpa\_num': 331, 'aportado\_cant': 5, 'provincia': 'P', 'cpa': 'DDCR12', 'tasa\_p\_mora': 12312.4312, 'tasa\_mora': 23123.12321, 'aglomerado': 'C', 'aportado\_fh\_num': '20171201','cod\_postal': 1400}, {'aportado\_fh': '2012-10', 'impacto\_codigo\_num': 12, 'geo\_nse': 'NA', 'pos': 1, 'avg\_tc\_limite\_credito': 1234.123, 'mes': 360, 'cpa\_num': 331, 'aportado\_cant': 4, 'provincia': 'P', 'cpa': 'DDCR12', 'tasa\_p\_mora': 12312.4312, 'tasa\_mora': 23123.12321, 'aglomerado': 'C', 'aportado\_fh\_num': '20180101', 'cod\_postal': 1406}, {'aportado\_fh': '2017-10', 'impacto\_codigo\_num': 12, 'geo\_nse': 'NA', 'pos': 1, 'avg\_tc\_limite\_credito': 1234.123, 'mes': 180, 'cpa\_num': 331, 'aportado\_cant': 2, 'provincia': 'P', 'cpa': 'DDCR12', 'tasa\_p\_mora': 12312.4312, 'tasa\_mora': 23123.12321, 'aglomerado': 'C', 'aportado\_fh\_num': '20180102', 'cod\_postal': 1407}]

Obteniendo el primero valor de la lista y accediento por su clave a cpa:

print Lista\_ordenada[0]['cpa']

-----------

DDCR12

En el ejemplo anterior se colocó en la variable pos, en todos los diccionarios, el valor 1, ya que este sería nuestro campo con mayor peso y necesitamos que ordene por el segundo en peso.

Modificando los valores de pos a 1, 2 y 3 (a los diccionarios \_1, \_2 y \_3) nos quedaría:

Lista\_ordenada = [{'aportado\_fh': '2012-10', 'impacto\_codigo\_num': 12, 'geo\_nse': 'NA', 'pos': 1, 'avg\_tc\_limite\_credito': 1234.123, 'mes': 360, 'cpa\_num': 331, 'aport

ado\_cant': 4, 'provincia': 'P', 'cpa': 'DDCR12', 'tasa\_p\_mora': 12312.4312, 'tasa\_mora': 23123.12321, 'aglomerado': 'C', 'aportado\_fh\_num': '20180101'

, 'cod\_postal': 1406}, {'aportado\_fh': '2017-10', 'impacto\_codigo\_num': 12, 'geo\_nse': 'NA', 'pos': 2, 'avg\_tc\_limite\_credito': 1234.123, 'mes': 180,

'cpa\_num': 331, 'aportado\_cant': 2, 'provincia': 'P', 'cpa': 'DDCR12', 'tasa\_p\_mora': 12312.4312, 'tasa\_mora': 23123.12321, 'aglomerado': 'C', 'aporta

do\_fh\_num': '20180102', 'cod\_postal': 1407}, {'aportado\_fh': '2017-09', 'impacto\_codigo\_num': 12, 'geo\_nse': 'NA', 'pos': 3, 'avg\_tc\_limite\_credito':

1234.123, 'mes': 60, 'cpa\_num': 331, 'aportado\_cant': 5, 'provincia': 'P', 'cpa': 'DDCR12', 'tasa\_p\_mora': 12312.4312, 'tasa\_mora': 23123.12321, 'aglo

merado': 'C', 'aportado\_fh\_num': '20171201', 'cod\_postal': 1400}]

En donde el diccionario con el menor valor de pos, nos queda primero en la lista.

* **Clases en Python**

Python permite manejar clases y crear objetos con métodos y atributos.  
Dentro de nuestra experiencia, para el desarrollo llevado a cabo, en una primera instancia se crearon alrededor de 17 clases (cada una representaba una fuente de datos), cada una contaba con entre 1 y 20 metodos. Estas clases eran instanciadas desde un proceso main (orquestador), para luego hacer uso de sus métodos.

Con el tiempo nos dimos cuenta que a Python le cuesta mucho llevar la referencia de tantos objetos y métodos y se vuelve muy lento en su procesamiento.

En una segunda instancia se eliminaron todas las clases y se las trabajaron funciones, con sub-funciones dentro de cada una. Comunicándose entre si haciendo uso de diccionarios y listas. Todo esto dentro de la misma función main (orquestador).

El resultado fue que, para procesar 17 fuentes de datos de alrededor de 200 mil registros (todas las fuentes residen en un mismo row, y cada fuente tiene estructura de datos complejas), el proceso tardaba alrededor de 12 horas.  
Luego de eliminar las clases, el mismo proceso comenzó a tardar menos de 1 hora.

En una última instancia, se decidió aislar este nuevo código sin clases, por cada fuente de datos, en una librería separada, de esta forma solo teníamos que invocar dicha librería y ejecutar la función que activaría el flujo de datos.

Ejemplo:

################ CON CLASES ####################

#\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Clase Consulta\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*#

class consulta\_class:

#Definicion Atributos#

ob\_list\_con\_matrices = []

ob\_con\_rub\_no\_f\_0a12 = int()

ob\_con\_srub\_fi1\_0a12 = int()

ob\_con\_rub\_no\_f\_0a24 = int()

ob\_con\_srub\_tc2\_0a24 = int()

ob\_con\_srub\_tec\_0a24 = int()

ob\_con\_cant\_consultas\_0a3 = int()

ob\_con\_srub\_tec\_0a3 = int()

ob\_con\_list\_srub\_0a60 = []

ob\_con\_rub\_no\_f\_0a60 = int()

ob\_con\_srub\_tc2\_0a60 = int()

ob\_con\_srub\_tec\_0a60 = int()

ob\_con\_cant\_consultas\_0a6 = int()

ob\_con\_rub\_f\_0a6 = int()

ob\_con\_rub\_no\_f\_0a6 = int()

ob\_con\_srub\_fi1\_0a6 = int()

ob\_con\_srub\_fi2\_0a6 = int()

ob\_con\_list\_meses\_rub\_f = []

ob\_con\_list\_meses\_srub\_fi1 = []

ob\_con\_list\_meses\_srub\_fi2 = []

ob\_con\_list\_meses\_srub\_tc2 = []

ob\_con\_list\_meses\_srub\_tec = []

#Metodos#

#----------get's-------------#

def get\_dif\_meses(self,fecha):

return diff\_month(par\_fecha\_aprobacion, fecha)

def get\_list\_con\_matrices(self):

return self.ob\_list\_con\_matrices

def get\_con\_rub\_no\_f\_0a12(self):

return self.ob\_con\_rub\_no\_f\_0a12

def get\_con\_srub\_fi1\_0a12(self):

return self.ob\_con\_srub\_fi1\_0a12

def get\_con\_rub\_no\_f\_0a24(self):

return self.ob\_con\_rub\_no\_f\_0a24

def get\_con\_srub\_tc2\_0a24(self):

return self.ob\_con\_srub\_tc2\_0a24

def get\_con\_srub\_tec\_0a24(self):

return self.ob\_con\_srub\_tec\_0a24

def get\_con\_cant\_consultas\_0a3(self):

return self.ob\_con\_cant\_consultas\_0a3

def get\_con\_srub\_tec\_0a3(self):

return self.ob\_con\_srub\_tec\_0a3

def get\_con\_list\_srub\_0a60(self):

return self.ob\_con\_list\_srub\_0a60

def get\_con\_rub\_no\_f\_0a60(self):

return self.ob\_con\_rub\_no\_f\_0a60

def get\_con\_srub\_tc2\_0a60(self):

return self.ob\_con\_srub\_tc2\_0a60

def get\_con\_srub\_tec\_0a60(self):

return self.ob\_con\_srub\_tec\_0a60

def get\_con\_cant\_consultas\_0a6(self):

return self.ob\_con\_cant\_consultas\_0a6

def get\_con\_rub\_f\_0a6(self):

return self.ob\_con\_rub\_f\_0a6

def get\_con\_rub\_no\_f\_0a6(self):

return self.ob\_con\_rub\_no\_f\_0a6

def get\_con\_srub\_fi1\_0a6(self):

return self.ob\_con\_srub\_fi1\_0a6

def get\_con\_srub\_fi2\_0a6(self):

return self.ob\_con\_srub\_fi2\_0a6

def get\_con\_list\_meses\_rub\_f(self):

return empty\_list(self.ob\_con\_list\_meses\_rub\_f)

def get\_con\_list\_meses\_srub\_fi1(self):

return empty\_list(self.ob\_con\_list\_meses\_srub\_fi1)

def get\_con\_list\_meses\_srub\_fi2(self):

return empty\_list(self.ob\_con\_list\_meses\_srub\_fi2)

def get\_con\_list\_meses\_srub\_tc2(self):

return empty\_list(self.ob\_con\_list\_meses\_srub\_tc2)

def get\_con\_list\_meses\_srub\_tec(self):

return empty\_list(self.ob\_con\_list\_meses\_srub\_tec)

def inic\_variables(self):

self.ob\_list\_con\_matrices = []

self.ob\_con\_rub\_no\_f = int()

self.ob\_con\_rub\_fi1 = int()

self.ob\_con\_rub\_no\_f\_0a24 = int()

self.ob\_con\_rub\_tc2\_0a24 = int()

self.ob\_con\_rub\_tec\_0a24 = int()

self.ob\_con\_cant\_consultas\_0a3 = int()

self.ob\_con\_rub\_tec\_0a3 = int()

self.ob\_con\_list\_srub\_0a60 = []

self.ob\_con\_rub\_no\_f\_0a60 = int()

self.ob\_con\_srub\_tc2\_0a60 = int()

self.ob\_con\_srub\_tec\_0a60 = int()

self.ob\_con\_cant\_consultas\_0a6 = int()

self.ob\_con\_rub\_f\_0a6 = int()

self.ob\_con\_rub\_no\_f\_0a6 = int()

self.ob\_con\_srub\_fi1\_0a6 = int()

self.ob\_con\_srub\_fi2\_0a6 = int()

self.ob\_con\_list\_meses\_rub\_f = []

self.ob\_con\_list\_meses\_srub\_fi1 = []

self.ob\_con\_list\_meses\_srub\_fi2 = []

self.ob\_con\_list\_meses\_srub\_tc2 = []

self.ob\_con\_list\_meses\_srub\_tec = []

def datasource\_consultas(self,consulta):

if hasattr(consulta,'PERIODO'):

meses = self.get\_dif\_meses(format\_date(NVL(consulta.PERIODO,'9999-01-01')))

else:

meses = self.get\_dif\_meses(format\_date('9999-01-01'))

#Rundate para retro

if 0 <= self.get\_dif\_meses(format\_date(NVL(consulta.INGRESO\_FH,'01/01/9999'))) <= 60:

#Condiciones

if (NVL(consulta.CONS\_TIPO,None) == 'C') and not(consulta.DUPLICADO) and NVL(consulta.PROD\_COD,'PROD\_COD') not in 'RISCPER' and (NVL(consulta.VISIBLE,False) == True) and (NVL(consulta.MATRIZ["TIPO"],'-') == 'A'):

#iq\_meses\_rub\_f

if (NVL(consulta.MATRIZ["RUBRO\_COD"],'-') == 'F'):

self.ob\_con\_list\_meses\_rub\_f.extend([meses])

#iq\_meses\_subrub\_fi1

if (NVL(consulta.MATRIZ["SUBRUBRO\_COD"],'-') == 'FI1'):

self.ob\_con\_list\_meses\_srub\_fi1.extend([meses])

#iq\_meses\_subrub\_fi2

if (NVL(consulta.MATRIZ["SUBRUBRO\_COD"],'-') == 'FI2'):

self.ob\_con\_list\_meses\_srub\_fi2.extend([meses])

#iq\_meses\_subrub\_tc2

if (NVL(consulta.MATRIZ["SUBRUBRO\_COD"],'-') == 'TC2'):

self.ob\_con\_list\_meses\_srub\_tc2.extend([meses])

#iq\_meses\_subrub\_tec

if (NVL(consulta.MATRIZ["SUBRUBRO\_COD"],'-') == 'TEC'):

self.ob\_con\_list\_meses\_srub\_tec.extend([meses])

#Periodo <= 3

if meses <= 3:

#iq\_0a3\_all

self.ob\_con\_cant\_consultas\_0a3 += 1

#iq\_0a3\_subrub\_tec

if (NVL(consulta.MATRIZ["SUBRUBRO\_COD"],'-') == 'TEC'):

self.ob\_con\_srub\_tec\_0a3 += 1

#Periodo <= 6

if meses <= 6:

#iq\_0a6\_mm

self.ob\_con\_cant\_consultas\_0a6 += 1

#iq\_0a6\_rub\_f

if (NVL(consulta.MATRIZ["RUBRO\_COD"],'-') == 'F'):

self.ob\_con\_rub\_f\_0a6 += 1

#iq\_0a6\_rub\_no\_f

if (NVL(consulta.MATRIZ["RUBRO\_COD"],'-') != 'F'):

self.ob\_con\_rub\_no\_f\_0a6 += 1

#iq\_0a6\_subrub\_fi1

if (NVL(consulta.MATRIZ["SUBRUBRO\_COD"],'-') == 'FI1'):

self.ob\_con\_srub\_fi1\_0a6 += 1

#iq\_0a6\_subrub\_fi2

if (NVL(consulta.MATRIZ["SUBRUBRO\_COD"],'-') == 'FI2'):

self.ob\_con\_srub\_fi2\_0a6 += 1

#Periodo <= 12

if meses <= 12:

#iq\_0a12\_mm

if ((NVL(consulta.MATRIZ["SUBRUBRO\_COD"],'-') == 'TEC') or (NVL(consulta.MATRIZ["SUBRUBRO\_COD"],'-') == 'TEF')) and consulta.MATRIZ["MATRIZ"]:

self.ob\_list\_con\_matrices.extend([consulta.MATRIZ["MATRIZ"]])

#iq\_0a12\_rub\_no\_f

if (NVL(consulta.MATRIZ["RUBRO\_COD"],'-') != 'F'):

self.ob\_con\_rub\_no\_f\_0a12 += 1

#iq\_0a12\_subrub\_fi1

if (NVL(consulta.MATRIZ["SUBRUBRO\_COD"],'-') == 'FI1'):

self.ob\_con\_srub\_fi1\_0a12 += 1

#Periodo <= 24

if meses <= 24:

#iq\_0a24\_rub\_no\_f

if (NVL(consulta.MATRIZ["RUBRO\_COD"],'-') != 'F'):

self.ob\_con\_rub\_no\_f\_0a24 += 1

#iq\_0a24\_subrub\_tc2

if (NVL(consulta.MATRIZ["SUBRUBRO\_COD"],'-') == 'TC2'):

self.ob\_con\_srub\_tc2\_0a24 += 1

#iq\_0a24\_subrub\_tec

if (NVL(consulta.MATRIZ["SUBRUBRO\_COD"],'-') == 'TEC'):

self.ob\_con\_srub\_tec\_0a24 += 1

#Periodo <= 60

if meses <= 60:

#iq\_0a59\_mr

self.ob\_con\_list\_srub\_0a60.extend([consulta.MATRIZ["SUBRUBRO\_COD"]])

#iq\_0a59\_rub\_no\_f

if (NVL(consulta.MATRIZ["RUBRO\_COD"],'-') != 'F'):

self.ob\_con\_rub\_no\_f\_0a60 += 1

#iq\_0a59\_subrub\_tc2

if (NVL(consulta.MATRIZ["SUBRUBRO\_COD"],'-') == 'TC2'):

self.ob\_con\_srub\_tc2\_0a60 += 1

#iq\_0a59\_subrub\_tec

if (NVL(consulta.MATRIZ["SUBRUBRO\_COD"],'-') == 'TEC'):

self.ob\_con\_srub\_tec\_0a60 += 1

-----------------  
En el main():

def auxConsultas(consulta):

ob\_consulta = consulta\_class()

ob\_consulta.inic\_variables()

ob\_consulta.datasource\_consultas(consulta)

consultas.append(ob\_consulta)

map(auxConsultas, row['CONSULTAS'])

################ SIN CLASES ####################

def c\_class(c):

#datasource\_consultas

def datasource\_consultas(consulta):

matriz = consulta.MATRIZ['MATRIZ']

if hasattr(consulta,'PERIODO'):

meses = get\_dif\_meses(format\_date(NVL(consulta.INGRESO\_FH,'9999-01-01')))

else:

meses = get\_dif\_meses(format\_date('9999-01-01'))

#Rundate para retro

if 0.0 <= meses <= 61.0:

#Condiciones

if (NVL(consulta.CONS\_TIPO,None) == 'C') and not(consulta.DUPLICADO) and consulta.PROD\_COD.find('RISCPER') < 0 and (NVL(consulta.VISIBLE,False)) and (NVL(consulta.MATRIZ["TIPO"],'-') == 'A'):

#iq\_meses\_rub\_f

if (NVL(consulta.MATRIZ["RUBRO\_COD"],'-') == 'F'):

variables\_consultas['ob\_con\_list\_meses\_rub\_f'].extend([meses])

#iq\_meses\_subrub\_fi1

if (NVL(consulta.MATRIZ["SUBRUBRO\_COD"],'-') == 'FI1'):

variables\_consultas['ob\_con\_list\_meses\_srub\_fi1'].extend([meses])

#iq\_meses\_subrub\_fi2

if (NVL(consulta.MATRIZ["SUBRUBRO\_COD"],'-') == 'FI2'):

variables\_consultas['ob\_con\_list\_meses\_srub\_fi2'].extend([meses])

#iq\_meses\_subrub\_tc2

if (NVL(consulta.MATRIZ["SUBRUBRO\_COD"],'-') == 'TC2'):

variables\_consultas['ob\_con\_list\_meses\_srub\_tc2'].extend([meses])

#iq\_meses\_subrub\_tec

if (NVL(consulta.MATRIZ["SUBRUBRO\_COD"],'-') == 'TEC'):

variables\_consultas['ob\_con\_list\_meses\_srub\_tec'].extend([meses])

#Periodo <= 3

if meses <= 3:

#iq\_0a3\_all

variables\_consultas['ob\_con\_cant\_consultas\_0a3'] += 1

#iq\_0a3\_subrub\_tec

if (NVL(consulta.MATRIZ["SUBRUBRO\_COD"],'-') == 'TEC'):

variables\_consultas['ob\_con\_srub\_tec\_0a3'] += 1

#Periodo <= 6

if meses <= 6:

#iq\_0a6\_mm

variables\_consultas['ob\_con\_cant\_consultas\_0a6'].extend([matriz])

#iq\_0a6\_rub\_f

if (NVL(consulta.MATRIZ["RUBRO\_COD"],'-') == 'F'):

variables\_consultas['ob\_con\_rub\_f\_0a6'] += 1

#iq\_0a6\_rub\_no\_f

if (NVL(consulta.MATRIZ["RUBRO\_COD"],'-') != 'F'):

variables\_consultas['ob\_con\_rub\_no\_f\_0a6'] += 1

#iq\_0a6\_subrub\_fi1

if (NVL(consulta.MATRIZ["SUBRUBRO\_COD"],'-') == 'FI1'):

variables\_consultas['ob\_con\_srub\_fi1\_0a6'] += 1

#iq\_0a6\_subrub\_fi2

if (NVL(consulta.MATRIZ["SUBRUBRO\_COD"],'-') == 'FI2'):

variables\_consultas['ob\_con\_srub\_fi2\_0a6'] += 1

#Periodo <= 12

if meses <= 12:

#iq\_0a12\_mm

if ((NVL(consulta.MATRIZ["SUBRUBRO\_COD"],'-') == 'TEC') or (NVL(consulta.MATRIZ["SUBRUBRO\_COD"],'-') == 'TEF')) and consulta.MATRIZ["MATRIZ"]:

variables\_consultas['ob\_list\_con\_matrices'].extend([consulta.MATRIZ["MATRIZ"]])

#iq\_0a12\_rub\_no\_f

if (NVL(consulta.MATRIZ["RUBRO\_COD"],'-') != 'F'):

variables\_consultas['ob\_con\_rub\_no\_f\_0a12'] += 1

#iq\_0a12\_subrub\_fi1

if (NVL(consulta.MATRIZ["SUBRUBRO\_COD"],'-') == 'FI1'):

variables\_consultas['ob\_con\_srub\_fi1\_0a12'] += 1

#Periodo <= 24

if meses <= 24:

#iq\_0a24\_rub\_no\_f

if (NVL(consulta.MATRIZ["RUBRO\_COD"],'-') != 'F'):

variables\_consultas['ob\_con\_rub\_no\_f\_0a24'] += 1

#iq\_0a24\_subrub\_tc2

if (NVL(consulta.MATRIZ["SUBRUBRO\_COD"],'-') == 'TC2'):

variables\_consultas['ob\_con\_srub\_tc2\_0a24'] += 1

#iq\_0a24\_subrub\_tec

if (NVL(consulta.MATRIZ["SUBRUBRO\_COD"],'-') == 'TEC'):

variables\_consultas['ob\_con\_srub\_tec\_0a24'] += 1

#Periodo <= 60

if meses <= 60:

#iq\_0a59\_mr

if NVL(consulta.MATRIZ["SUBRUBRO\_COD"],'-') <> '-':

variables\_consultas['ob\_con\_list\_srub\_0a60'].extend([consulta.MATRIZ["SUBRUBRO\_COD"]])

#iq\_0a59\_rub\_no\_f

if (NVL(consulta.MATRIZ["RUBRO\_COD"],'-') != 'F'):

variables\_consultas['ob\_con\_rub\_no\_f\_0a60'] += 1

#iq\_0a59\_subrub\_tc2

if (NVL(consulta.MATRIZ["SUBRUBRO\_COD"],'-') == 'TC2'):

variables\_consultas['ob\_con\_srub\_tc2\_0a60'] += 1

#iq\_0a59\_subrub\_tec

if (NVL(consulta.MATRIZ["SUBRUBRO\_COD"],'-') == 'TEC'):

variables\_consultas['ob\_con\_srub\_tec\_0a60'] += 1

datasource\_consultas(c)

#DATASOURCE CONSULTAS#

#DEFINICION DE VARIABLES DEL DICCIONARIO#

variables\_consultas = {}

variables\_consultas['ob\_con\_list\_meses\_rub\_f'] = []

variables\_consultas['ob\_con\_list\_meses\_srub\_fi1'] = []

variables\_consultas['ob\_con\_list\_meses\_srub\_fi2'] = []

variables\_consultas['ob\_con\_list\_meses\_srub\_tc2'] = []

variables\_consultas['ob\_con\_list\_meses\_srub\_tec'] = []

variables\_consultas['ob\_con\_cant\_consultas\_0a3'] = 0

variables\_consultas['ob\_con\_srub\_tec\_0a3'] = 0

variables\_consultas['ob\_con\_cant\_consultas\_0a6'] = []

variables\_consultas['ob\_con\_rub\_f\_0a6'] = 0

variables\_consultas['ob\_con\_rub\_no\_f\_0a6'] = 0

variables\_consultas['ob\_con\_srub\_fi1\_0a6'] = 0

variables\_consultas['ob\_con\_srub\_fi2\_0a6'] = 0

variables\_consultas['ob\_list\_con\_matrices'] = []

variables\_consultas['ob\_con\_rub\_no\_f\_0a12'] = 0

variables\_consultas['ob\_con\_srub\_fi1\_0a12'] = 0

variables\_consultas['ob\_con\_rub\_no\_f\_0a24'] = 0

variables\_consultas['ob\_con\_srub\_tc2\_0a24'] = 0

variables\_consultas['ob\_con\_srub\_tec\_0a24'] = 0

variables\_consultas['ob\_con\_list\_srub\_0a60'] = []

variables\_consultas['ob\_con\_rub\_no\_f\_0a60'] = 0

variables\_consultas['ob\_con\_srub\_tc2\_0a60'] = 0

variables\_consultas['ob\_con\_srub\_tec\_0a60'] = 0

#HACEMOS EL DATASOURCE#

map(c\_class,row.CONSULTAS)