Programación para el Análisis de Datos

Clase 1 - Fundamentos de programación en Python

Referencias y bibliografía de consulta:

- Python for Data Analysis by Wes McKinney (O'Reilly) 2018
- Documentación oficial de Python (https://docs.python.org/)
- PyPi (https://pypi.org/)

Sobre Python

Python es un lenguaje de programación de alto nivel y de propósito general lanzado por Guido van Rossum en 1991. Actualmente es uno de los lenguajes de programación más utilizados principalmente por su **simpleza** y **versatilidad**.

Ventajas

- Facilidad de uso
- Popularidad
- Open Source
- Integración con aplicaciones empresariales

Desventajas

- Ejecución lenta por ser un lenguaje interpretado

Instalación de paquetes

Para instalar un módulo en python se utilizan

```
# PIP
pip install $MODULE
# Anaconda
conda install $MODULE
```

En un notebook de jupyter se puede utilizar el *magic command* **!** para ejecutar un comando por consola

```
!pwd
```

In [18]:
```
!cd
```

```
C:\Users\jzaff
```

1 - Introducción

1.1 - Tabulaciones

A diferencia de otros lenguajes de programación que utilizan símbolos (*{ }*, *[ ]*), Python utiliza tabulaciones o espacios para delimitar y estructurar los bloques de código.

In [17]:
```
lista = [1,3,35,5,6,7,8]

for i in lista:
    if i % 2 == 0:
        print("{} es par".format(i))
    else:
        print("{} es inpar".format(i))
```

```
1 es inpar
3 es inpar
35 es inpar
5 es inpar
6 es par
7 es inpar
8 es par
```

1.2 - Comentarios

Cualquier texto precedido por la marca hash (signo de numeración) # es ignorado por el intérprete de Python. Esto se usa a menudo para añadir comentarios al código. A veces también puede querer excluir ciertos bloques de código sin borrarlos.

In [3]:
```
# Creamos una lista de números:
lista = [1,2,3,4,5,6,7,8]

# Iteramos por los elementos de la lista
for i in lista:
    if i % 2 == 0: # Si el numero es par
        print("{} es par".format(i))
    else:
        print("{} es inpar".format(i))
```

```
1 es inpar
2 es par
3 es inpar
4 es par
5 es inpar
6 es par
7 es inpar
8 es par
```

1.3 - Función help()

Esta función se utiliza para mostrar una breve descripción de uso de una función específica. Si se utiliza la función sin argumentos, se inicializa una consola de ayuda.

In [4]:
```
help('print')
```

```
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdou
t.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

1.4 - Tipado dinámico

Cuando se asigna una variable (o nombre) en Python, se está creando una referencia al objeto en el lado derecho del signo igual.

Este objeto no tiene un tipo de dato fijo. El tipo de dato depende de qué se está almacenando en la variable.

In [5]:
```
var = 1
print(type(var))
```

```
<class 'int'>
```

In [6]:
```
var = 2.0
print(type(var))
```

```
<class 'float'>
```

In [7]:
```
var = 'a'
print(type(var))
```

```
<class 'str'>
```

In [8]:
```
var = [1, 2]
print(type(var))
```

```
<class 'list'>
```

In [9]:
```
var = (1, 2)
print(type(var))
```

```
<class 'tuple'>
```

Las variables son nombres de objetos dentro de un determinado espacio de nombres; la información de tipo se almacena en el propio objeto.

Pero no podemos utilizar cualquier operación con cualquier tipo de dato.

En el siguiente ejemplo se muestra una operación no permitida:

In [10]:
```
try:
    5 + '5'
except Exception as e:
    print("Error:", e)
```

```
Error: unsupported operand type(s) for +: 'int' and 'str'
```

En este sentido, se considera que Python es un lenguaje fuertemente tipificado, lo que significa que cada objeto tiene un tipo (o clase) específico, y las conversiones implícitas sólo se producirán en determinadas circunstancias evidentes, como se puede ver en los siguientes ejemplos:

In [11]:
```
a = 4.5
b = 2

print('a is {0}, b is {1}'.format(type(a), type(b)))
print('a/b= ', a/b)
```

```
a is <class 'float'>, b is <class 'int'>
a/b=  2.25
```

En el caso de utilizar una función con el tipo de dato erroneo, se produce una excepción. Una Excepción es un evento que se produce ante una situación que el interprete no puede resolver.

Para mas información sobre el tema se puede visitar el siguiente link (https://docs.python.org/3.8/tutorial/errors.html#exceptions)

2 - Tipos de dato

2.1 - Numéricos

In [12]:
```
# Entero
num1 = 3
print("num1 es del tipo", type(num1))
```

```
num1 es del tipo <class 'int'>
```

In [13]:
```
# Real
num2 = 3.1
print("num2 es del tipo", type(num2))
```

```
num2 es del tipo <class 'float'>
```

In [14]:
```
## Numeros complejos
c1 = 1 - 1j
c2 = complex(real=-2, imag=2)

print("c1 es del tipo", type(c1))
print("c2 es del tipo", type(c2))
```

```
c1 es del tipo <class 'complex'>
c2 es del tipo <class 'complex'>
```

In [15]:
```
c1
```

Out[15]:
```
(1-1j)
```

In [16]:
```
c2
```

Out[16]:
```
(-2+2j)
```

Operaciones básicas

In [17]:
```
print("Suma: num1 + num2 =", num1 + num2)
```

```
Suma: num1 + num2 = 6.1
```

In [18]:
```
print("Resta: num1 - num2 = {:.2f}".format(num1 - num2))
```

```
Resta: num1 - num2 = -0.10
```

In [19]:
```
print("Negacion: -num1 =", -num1)
```

```
Negacion: -num1 = -3
```

In [20]:
```
print("Multiplicación: num1 x 2 =", num1*2)
```

```
Multiplicación: num1 x 2 = 6
```

In [21]:
```
print("División: num1/num2 = {:.2f}".format(num1/num2))
```

```
División: num1/num2 = 0.97
```

In [22]:
```
print("División entera: 3 // 2 =", 3 // 2)
```

```
División entera: 3 // 2 = 1
```

In [23]:
```
print("Módulo: num1 % 2 =", num1 % 2)
```

```
Módulo: num1 % 2 = 1
```

In [24]:
```
print("Exponenciación: num1 ** 2 =", num1 ** 2)
```

```
Exponenciación: num1 ** 2 = 9
```

2.2 - String

In [25]:
```
docstring = """Docstring
                Se puede utilizar para realizar comentarios multi-linea"""

string1 = "Se pueden incluir comillas 'simples'."
string2 = 'tercera forma de definir string'
```

In [26]:
```
print(docstring)
```

```
Docstring
                Se puede utilizar para realizar comentarios multi-linea
```

Operaciones básicas

Los strings se almacenan como listas y pueden ser accedidos mediante indices.

NOTA: Los indices en Python inician en **0**

In [27]:
```
print('La primera componente de string1 es:', string1[0])
```

```
La primera componente de string1 es: S
```

Definamos 2 strings:

In [28]:
```
a = "Hello"
b = "World"
```

In [29]:
```
# Operador SUMA
print('La concatenación entre {} y {} es: {}'.format(a, b, a+b))
```

```
La concatenación entre Hello y World es: HelloWorld
```

In [30]:
```
# Función join
' '.join(['Hello','World'])
```

Out[30]: 'Hello World'

In [31]:
```
# Largo del string
print("El string '{}' tiene un largo de {} caracteres".format(a, len(a)))
```

```
El string 'Hello' tiene un largo de 5 caracteres
```

In [32]:
```
# Eliminacion de espacios
print("  Este string   tiene espacios    ".strip())
print("### Tambien se pueden eliminar caracteres especiales $%".strip('# $%
```

```
Este string   tiene espacios
Tambien se pueden eliminar caracteres especiales
```

In [33]:
```
# Dividir un string
print("Esto,es,el,registro,de,un,csv".split(','))
```

```
['Esto', 'es', 'el', 'registro', 'de', 'un', 'csv']
```

In [34]:
```
# UPPERCASE y lowercase
print("Este string tiene minusculas".upper())
print("ESTE SRING ESTA EN MAYUSCULA".lower())
```

```
ESTE STRING TIENE MINUSCULAS
este sring esta en mayuscula
```

3 - Operadores

Un operador es un símbolo que realiza una operación.

Existen distintos tipos de operadores que se pueden agrupar de la siguiente forma: - Operadores aritméticos - Operadores relacionales - Operadores de asignación - Operadores lógicos - Operadores de pertenencia - Operador binario

3.1 - Operadores aritméticos:

In [35]:
```
# Suma
print("Suma: 1 + 2 = {}".format(1+2))
# Resta
print("Resta: 1 - 2 = {}".format(1-2))
# División
print("División: 10 / 3 = {}".format(10/3))
# División entera
print("División entera: 3 // 2 = {}".format(3//2))
# Módulo
print("Módulo / Resto : 257 % 2 = {}".format(257 % 2))
# Multiplicación
print("Multiplicación: 7 * 8 = {}".format(7 * 8))
# Exponenciación
print("Exponenciación 2**3 = {}".format(2 ** 3))
```

```
Suma: 1 + 2 = 3
Resta: 1 - 2 = -1
División: 10 / 3 = 3.3333333333333335
División entera: 3 // 2 = 1
Módulo / Resto : 257 % 2 = 1
Multiplicación: 7 * 8 = 56
Exponenciación 2**3 = 8
```

3.2 - Operadores relacionales

Relacionan 2 variables y retornan una variable del tipo *booleana*

In [36]:
```
a = 10
# Mayor
print("a > 3.14 = {}".format(a > 3.14))
# Menor
print("a < 3.14 = {}".format(a < 3.14))
# Mayor o igual
print("a >= 3.14 = {}".format(a >= 3.14))
# Menor o igual
print("a <= 3.14 = {}".format(a <= 3.14))
# Distinto
print("a != 3.14 = {}".format(a != 3.14))
# Igual
print("a == 3.14 = {}".format(a == 3.14))

# Las variables booleanas se representan numericamente como 0 (falso) y 1 (
print(0 == False)
print(1 == True)
```

```
a > 3.14 = True
a < 3.14 = False
a >= 3.14 = True
a <= 3.14 = False
a != 3.14 = True
a == 3.14 = False
True
True
```

3.3 - Operadores de asignación

Estos operadores son utilizados para realizar asignaciones a variables:

Suma:

In [37]:
```
var = 2
var += 2
print("Suma y asignacion: var += 2 --> {}".format(var))
```

```
Suma y asignacion: var += 2 --> 4
```

In [38]:
```
var = 2
var = var + 2
print(var)
```

```
4
```

Resta:

In [39]:
```
var = 2
var -= 2
print("Resta y asignacion: var -= 2 --> {}".format(var))
```

```
Resta y asignacion: var -= 2 --> 0
```

División:

In [40]:
```
var = 2
var /= 3
print("División y asignacion: var /= 3 --> {:.4f}".format(var))
```

```
División y asignacion: var /= 3 --> 0.6667
```

División entera:

In [41]:
```
var = 2
var //= 2
print("División entera y asignacion: var //= 2 --> {}".format(var))
```

```
División entera y asignacion: var //= 2 --> 1
```

Módulo:

In [42]:
```
var = 2
var  %=  2
print("Módulo / Resto  y asignación: var  %=  2 --> {}".format(var))
```

```
Módulo / Resto  y asignación: var  %=  2 --> 0
```

Multiplicación:

In [43]:
```
var = 2
var  *=  8
print("Multiplicación y asignación: var  *=  8 --> {}".format(var))
```

```
Multiplicación y asignación: var  *=  8 --> 16
```

Exponenciación:

In [44]:
```
var = 2
var  **=  3
print("Exponenciación y asignación: var  **=  3 --> {}".format(var))
```

```
Exponenciación y asignación: var  **=  3 --> 8
```

3.4 - Operadores lógicos

Los operadores lógicos booleanos permiten agregar una lógica mas compleja a los programas

- and

In1	In2	Out
False	False	False
False	True	False
True	False	False
True	True	True

- or

In1	In2	Out
False	False	False
False	True	True
True	False	True
True	True	True

- not

In1	Out
False	True
True	False

AND:

In [45]:
```
1 > 2 and 3 == 3
```

Out[45]: False

OR:

In [46]:
```
1 > 2 or 3 == 3
```

Out[46]: True

NOT:

In [47]:
```
var = 2
not var == 2
```

Out[47]: False

3.5 - Operadores de pertenencia

Estos operadores se utilizan para determinar si un elemento se encuentra presente en una sequencia.

In:

```
In [48]: 1 in [ 4, 6, 1, 3, 0, 'list', 3.1]
Out[48]: True
```

```
In [49]: 5 in [ 4, 6, 1, 3, 0, 'list', 3.1]
Out[49]: False
```

Este operador tambien se puede utilizar para evaluar pertenencia en un string:

```
In [50]: 'Python' in 'Esta es una introducción a Python'
Out[50]: True
```

Not in:

```
In [51]: 1 not in [ 4, 6, 1, 3, 0, 'list', 3.1]
Out[51]: False
```

Precaución con la precedencia de los operadores

Operators	Meaning
()	Parentheses
**	Exponent
+x, -x, ~x	Unary plus, Unary minus, Bitwise NOT
*, /, //, %	Multiplication, Division, Floor division, Modulus
+, -	Addition, Subtraction
<<, >>	Bitwise shift operators
&	Bitwise AND
^	Bitwise XOR
|	Bitwise OR
==, !=, >, >=, <, <=, is, is not, in, not in	Comparisons, Identity, Membership operators
not	Logical NOT
and	Logical AND
or	Logical OR

4 - Estructuras

4.1 - Tuplas

Las tuplas son secuencias:

- ordenadas
- inmutables
- permiten elementos duplicados

Una variable inmutable no puede ser modificada. Es decir que son estáticos.

In [52]:
```
tupla = (1 , 2 , 3 , 4, 5)

print("La variable de tipo {} contiene los siguientes elementos: {}".format
```

```
La variable de tipo <class 'tuple'> contiene los siguientes elementos:
(1, 2, 3, 4, 5)
```

Para definir una tupla con un unico componente, es necesario definirlo de la siguiente manera:

In [53]:
```
t1 = (1)
t2 = (1, )
print(type(t1))
print(type(t2))
```

```
<class 'int'>
<class 'tuple'>
```

Acceso a los datos:

In [54]:
```
print("El primer componente de la tupla es:", tupla[0])
```

```
El primer componente de la tupla es: 1
```

Se puede utilizar un indice negativo para contar desde el final de la lista:

In [55]:
```
print("El último componente de la tupla es:", tupla[-1])
```

```
El último componente de la tupla es: 5
```

Una forma muy utilizada de acceder a datos se conoce como *slicing*. Consiste en determinar la posición inicial y final de la sublista de la siguiente forma:

```
tupla[desde:hasta+1:paso]
```

In [56]:
```
print("Desde el indice 3 hasta el indice 4:", tupla[3:5])
```

```
Desde el indice 3 hasta el indice 4: (4, 5)
```

In [57]:
```
print("Desde el principio hasta la ante última posición con un paso de 2 el
print(tupla[:-1:2])
```

```
Desde el principio hasta la ante última posición con un paso de 2 element
os:
(1, 3)
```

Las tuplas no permiten asignación:

In [58]:
```
try:
    tupla[2] = 3
except Exception as e:
    print(e)
```

```
'tuple' object does not support item assignment
```

4.2 - Listas

Una lista es una estructura similar a las tuplas, con las siguientes características:

- ordenada
- mutable
- permite elementos duplicados.

La lista posee el mismo manejo de datos que las tuplas, pero se le agrega la posibilidad de asignación y modificación de elementos.

Definición

Las listas se pueden definir de varias maneras. La más usual es:

In [59]:
```
lista = [1 , 2 , 3 , 4, 5]
lista
```

Out[59]:
```
[1, 2, 3, 4, 5]
```

Se puede utilizar el constructor `list()` :

In [60]:
```
l = list((1, 2))
print("l es una variable de tipo {}, que contiene los elementos: {}".format
```

```
l es una variable de tipo <class 'list'>, que contiene los elementos: [1,
2]
```

Acceso a los datos:

In [61]:
```
print("El segundo componente de la lista es:", lista[1])
```

```
El segundo componente de la lista es: 2
```

Se puede utilizar un indice negativo para contar desde el final de la lista:

In [62]:
```
print("El último componente de la lista es:", lista[-1])
```

```
El último componente de la lista es: 5
```

Accedamos a los datos de la lista usando *slicing*.

```
lista[desde:hasta+1:paso]
```

In [63]:
```
print("Desde el indice 2 hasta el indice 3: {}".format(lista[2:4]))
print("Desde el principio hasta la ante última posición: {}".format(lista[:
```

```
Desde el indice 2 hasta el indice 3: [3, 4]
Desde el principio hasta la ante última posición: [1, 2, 3, 4]
```

Python permite utilizar operadores de pertenencia sobre listas. Esta es la forma más usual de realizar un control de flujo.

In [64]:
```
l = ["el1", "el2", "el3", "el4", "el5", "el6", "el7", "el8"]
el = "el2"
if el in l:
    print("El elemento {} pertenece a la lista l".format(el))
else:
    print("El elemento {} no pertenece a la lista l".format(el))
```

```
El elemento el2 pertenece a la lista l
```

Manejo de datos

Las operaciones básicas de manejo de elementos son:

- agregar elementos
- eliminar elementos
- ordenar elementos

In [65]:
```
# Obtener elementos
l1 = list(range(1,11)) # Crea una lista de 10 elementos con numeros enteros
l1
```

Out[65]:
```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Pop

In [66]:
```
# Se muestra el 3er componente de la lista y se elimina:
print(l1.pop(2))
l1[:]
```

```
3
```

Out[66]:
```
[1, 2, 4, 5, 6, 7, 8, 9, 10]
```

In [67]:
```
# Se muestra el último componente de la lista y se elimina:
print(l1.pop())
l1
```

```
10
```

Out[67]:
```
[1, 2, 4, 5, 6, 7, 8, 9]
```

In [68]:
```
# Se muestra el ante-último componente de la lista y se elimina
print(l1.pop(-2))
l1
```

```
8
```

Out[68]:
```
[1, 2, 4, 5, 6, 7, 9]
```

Agregar y modificar elementos:

In [69]:
```
l1 = list(range(1,11))
l1
```

Out[69]:
```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

In [70]:
```
# Modificar el elemento en un indice:
l1[3] += 4
print("Lista: {}".format(l1))
```

```
Lista: [1, 2, 3, 8, 5, 6, 7, 8, 9, 10]
```

In [71]:
```
# No se pueden agregar elementos asignándolos, unicamente modificarlos:
try:
    l1[12] = 0
except Exception as e:
    print(e)
```

```
list assignment index out of range
```

In [72]:
```
# Insert
position = 2
element = "nuevo"
l1.insert(position, element)
l1
```

Out[72]:
```
[1, 2, 'nuevo', 3, 8, 5, 6, 7, 8, 9, 10]
```

In [73]:
```
# Si la posicion es mayor a len(list),
# se agrega el elemento en la ultima posicion de la lista:
position = 15
element = [1, 2]
l1.insert(position, element)
l1
```

Out[73]:
```
[1, 2, 'nuevo', 3, 8, 5, 6, 7, 8, 9, 10, [1, 2]]
```

In [74]:
```
# Append: agrega un único elemento al final de la lista:

element = "elemento appendeado"
l1.append(element)
l1
```

Out[74]:
```
[1, 2, 'nuevo', 3, 8, 5, 6, 7, 8, 9, 10, [1, 2], 'elemento appendeado']
```

In [75]:
```
# Extend: agrega multiples elementos al final de una lista.
elements = [1, 2]
l1.extend(elements)
l1
```

Out[75]:
```
[1, 2, 'nuevo', 3, 8, 5, 6, 7, 8, 9, 10, [1, 2], 'elemento appendeado',
1, 2]
```

Eliminacion de elementos

In [76]:
```
l1 = list(range(1,11))
```

In [77]:
```
l1
```

Out[77]:
```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

In [78]:
```
# remove
l1.remove(5)
l1
```

Out[78]:
```
[1, 2, 3, 4, 6, 7, 8, 9, 10]
```

In [79]:
```
# El elemento debe pertenecer a la lista
try:
    l1.remove("No Pertenece")
except Exception as e:
    print(e)
```

```
list.remove(x): x not in list
```

In [80]:
```
#  del
del l1[2]
l1
```

Out[80]:
```
[1, 2, 4, 6, 7, 8, 9, 10]
```

In [81]:
```
# Se pueden eliminar varios elementos utilizando slicing
del l1[2:5]
l1
```

Out[81]: [1, 2, 8, 9, 10]

Sort

In [82]:
```
listaSort = [2,4,3,1]

listaSort.sort()
listaSort
```

Out[82]: [1, 2, 3, 4]

4.3 - Diccionarios

Los diccionarios son estructuras que almacenan colecciones de tipo clave-valor (key-value)

Las principales características de los diccionarios son:

- Estructura tipo key-value
- Desordenado
- mutable

Los valores de las keys deben ser inmutables. Pueden ser tanto valores numéricos, strings, como tuplas.

Definición

In [83]:
```
d1 = {'key1': 'value1', 2:[ 1, 2, 3]}
print(d1)
```

```
{'key1': 'value1', 2: [1, 2, 3]}
```

Asignación:

In [84]:
```
d1['newKey'] = 3.1415
print(d1)
```

```
{'key1': 'value1', 2: [1, 2, 3], 'newKey': 3.1415}
```

Union de multiples diccionarios:

In [85]:
```
d2 = dict(zip(range(5), reversed(range(5))))

d1.update(d2)

print('d2: {}'.format(d2))
print('d1: {}'.format(d1))
```

```
d2: {0: 4, 1: 3, 2: 2, 3: 1, 4: 0}
d1: {'key1': 'value1', 2: 2, 'newKey': 3.1415, 0: 4, 1: 3, 3: 1, 4: 0}
```

In [86]:
```
# La función zip une los elementos de 2 listas en forma de tuplas.
# Es necesario que ambas listas tengan el mismo largo
l1 = [1, 2, 3]
l2 = [6, 5, 4]

for element in zip( l1, l2):
    print(element)
```

```
(1, 6)
(2, 5)
(3, 4)
```

Manejo de datos

El acceso a los elementos es similar al de las listas. Las claves se consideran como indices.

In [87]:
```
key = 'key1'
try:
    print("El valor almacenado en {} es: {}".format(key, d1[key]))
except Exception as e:
    print("No se encuentra {} en las keys: {}".format(key, d1.keys()))
```

```
El valor almacenado en key1 es: value1
```

La eliminación de registros se puede hacer mediante las funciones *del* y *pop*

In [88]:
```
d1['newElem'] = 'Elem'
print('d1:', d1)
r = d1.pop('newElem')
print('r:', r)
print('d1:', d1)
```

```
d1: {'key1': 'value1', 2: 2, 'newKey': 3.1415, 0: 4, 1: 3, 3: 1, 4: 0, 'n
ewElem': 'Elem'}
r: Elem
d1: {'key1': 'value1', 2: 2, 'newKey': 3.1415, 0: 4, 1: 3, 3: 1, 4: 0}
```

In [89]:
```
# En caso de no encontrar el elemento, se puede retornar un valor por defec

r = d1.get('KEY', "No Encontrado!")

print(r)
```

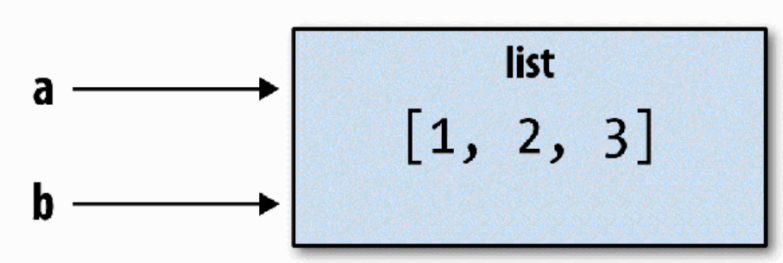
```
No Encontrado!
```

4.4 - Las variables en Python

Analicemos las siguientes líneas de código:

In [90]:
```
a = [1, 2, 3]
b = a
```

En algunos lenguajes, esta asignación provocaría que los datos [1, 2, 3] se copien. En Python, a y b se refieren ahora al mismo objeto, la lista original [1, 2, 3]



Podemos corroborar esto, agregando un elemento al final de la lista a y luego examinando b:

In [91]:
```
a.append(4)
b
```

Out[91]:
```
[1, 2, 3, 4]
```

In [92]:
```
a
```

Out[92]:
```
[1, 2, 3, 4]
```

copy y deepcopy

Para que una variable haga referencia a los datos y no a la memoria, tenemos que realizarlo de forma explicita al momento de crear la nueva variable con el método copy().

In [93]:
```
from copy import copy

c = copy(a)

a.append(5)

c
```

Out[93]:
```
[1, 2, 3, 4]
```

Existen situaciones donde las variables almacenan estructuras complejas. En esos casos `copy` unicamente realiza una copia de la variable y el contenido de la estructura la referencia.

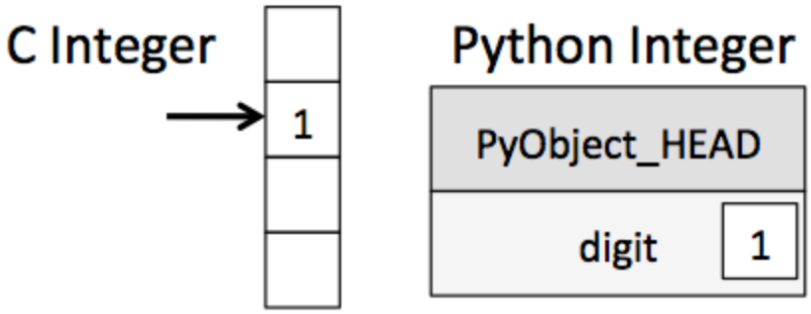
In [94]:
```
from copy import deepcopy

# initializamos la lista 1
l1 = [1, 2, [3,5], 4]

# Hacemos una copia de l1:
l2 = copy(l1)

# Hacemos una deep copy de l1:
l3 = deepcopy(l1)

# Modificamos algunos valores de l1:
l1[0] = 2
l1[2][0] = 6

# Mostramos por pantalla a las diferentes listas para evaluar los resultado
print("lista original : {}".format(l1))
print("lista copy : {}".format(l2))
print("lista deepcopy : {}".format(l3))
```

```
lista original : [2, 2, [6, 5], 4]
lista copy : [1, 2, [6, 5], 4]
lista deepcopy : [1, 2, [3, 5], 4]
```

Se puede ver que la lista creada con copy no varia los componentes de tipo simple, pero sí la lista almacenada en la posición 2.

Deepcopy no tiene este problema ya que realiza una copia recursiva de toda la estructura.

Los integer de Python:

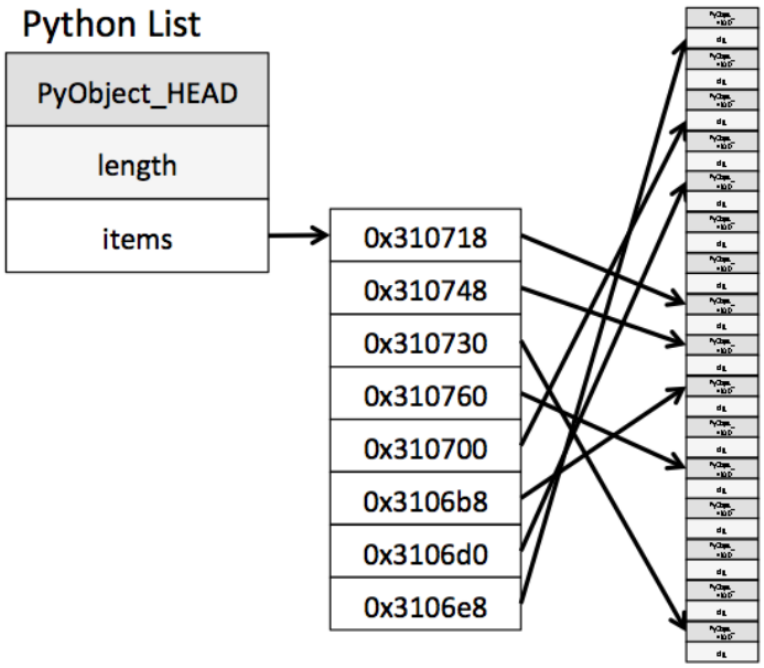


Cada objeto de Python es simplemente una estructura C que contiene no sólo su valor, sino también otra información. Por ejemplo el int 1, tendría la estructura:

```
struct _longobject {
    long ob_refcnt;
    PyTypeObject *ob_type;
    size_t ob_size;
    long ob_digit[1];
};
```

- `ob_refcnt` : un conteo de referencias que ayuda a Python a manejar silenciosamente la asignación y desasignación de memoria
- `ob_type` : codifica el tipo de la variable

Las listas de Python:



Para permitir estos tipos flexibles, cada elemento de la lista debe contener su propia información de tipo, número de referencias y otra información, es decir, cada elemento es un objeto Python completo.

La lista de Python contiene un puntero a un bloque de punteros, cada uno de los cuales apunta a su vez a un objeto completo de Python como el entero de Python que vimos antes. De nuevo, la ventaja de la lista es la flexibilidad: como cada elemento de la lista es una estructura

5 - Control de flujo

Python permite agregarle lógica a los programas y funciones mediante loops y condicionales

5.1 - Condicionales

Un condicional permite ejecutar distintas partes de código en función de valores **lógicos**.

if - else

Este condicional evalua el valor lógico de una sentencia y determina el código que se ejecuta.

In [95]:
```
boolean = True # False

if boolean:
    print("La sentencia es verdadera")
else:
    print("la sentencia es falsa")
```

```
La sentencia es verdadera
```

Es posible dividir el flujo en mas de dos ramas, utilizando la palabra clave **elif**

In [96]:
```
num = 20

if 4 < num < 10:
    print("opción 1")
elif num < 4:
    print("opcion 2")
else:
    print("opción por defecto")
```

```
opción por defecto
```

5.2 - Ciclos

Los ciclos permiten ejecutar bloques de código multiples veces, haciendo que el código sea legible y compacto.

for

Este ciclo realiza una cantidad fija de repeticiones. La cantidad de iteraciones es determinada por un iterador. Un iterador es un tipo de dato que posee multiples elementos por los cuales se puede iterar en un orden. Los ejemplos más comunes son repetir una candidad *N* de veces un bloque de código o para cada componente en una sequencia ejecutar ciertos comandos.

In [97]:
```
for i in range(5): # Se realizan 5 repeticiones
    print("Loop número {}.".format(i+1))
```

```
Loop número 1.
Loop número 2.
Loop número 3.
Loop número 4.
Loop número 5.
```

In [98]:
```
for fruit in ['Manzana', 'Banana', 'Naranja']: # Para cada elemento de la l
    print("La fruta seleccionada es: {}".format(fruit))
```

```
La fruta seleccionada es: Manzana
La fruta seleccionada es: Banana
La fruta seleccionada es: Naranja
```

while

Este bucle se ejecutará siempre y cuando se cumpla alguna condición.

In [99]:
```
i = 1
while i < 6:
    print(i)
    i += 1
else:
    print("Loop finalizado")
```

```
1
2
3
4
5
Loop finalizado
```

Tanto en los loops *for* y *while* existe la posibilidad de ejecutar un bloque de código luego de finalizar el bucle, utilizando la palabra clave **else**. Esto se suele utilizar para realizar tareas posteriores al loop. Por ejemplo en el bucle se leen las tablas de una base de datos (DB) y posteriormente se cierran las conexiones.

Condiciones de corte

Python permite más funcionalidades a la hora de realizar control de flujo. A continuación veremos el comportamiento de las keywords **break** y **continue**

- break: El interprete sale del bucle cuando lee esta palabra clave y continua con la ejecucion del programa.
- continue: El interprete intenta ejecutar el próximo bucle del ciclo

In [100]:
```
i = 1
while i < 6:
    print(i)
    if i == 3:
        break
    i += 1
```

```
1
2
3
```

In [101]:
```
i = 0
while i < 6:
    i += 1
    if i == 3:
        continue
    print(i)
```

```
1
2
4
5
6
```

Iteradores

Los iteradores de código son los objetos por los cuales se utilizan los bucles. Python da la posibilidad de crear iteradores personalizados. En esta clase se verá una leve introducción que se profundira en el proximo encuentro.

Para utilizar un iterador, es necesario conocer las funciones **iter()** y **next()**

- iter: Permite crear un iterador básico en función de un tipo de dato sequencial (lista, tupla, diccionario).
- next: Devuelve el siguiente elemento del iterador.

A continuación se muestra un ejemplo

In [102]:
```
tupla = ("Manzana", "Banana", "Naranja")
iterador = iter(tupla)

print(next(iterador))
print(next(iterador))
print(next(iterador))
```

```
Manzana
Banana
Naranja
```

6 - Funciones de secuencia incorporadas

6.1 - `Enumerate`

`enumerate` lleva el "registro" del índice de cada elemento de la lista.

La función `enumerate()` devuelve un objeto de la clase `enumerate` que contiene tuplas que aparejan cada elemento con su respectivo índice.

In [103]:
```
a = ['a','b','c','d']

enumerate(a)
```

Out[103]:
```
<enumerate at 0x105dab680>
```

In [104]:
```
enumerated = []

for i in enumerate(a):
    enumerated.append(i)

enumerated
```

Out[104]:
```
[(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd')]
```

Alternativa obteniendo por separado lo elementos de la tupla:

In [105]:
```
for i, e in enumerate(a):
    print('La letra en la posición {} es {}'.format(i+1,e))
```

```
La letra en la posición 1 es a
La letra en la posición 2 es b
La letra en la posición 3 es c
La letra en la posición 4 es d
```

Nota sobre unpacking de tuplas y listas:

In [106]:
```
tup = (1,2,3,4)
```

In [107]:
```
a, b, c, d = tup
```

In [108]:
```
a
```

Out[108]:
```
1
```

6.2 - `Zip`

`zip` recorre cada elemento de dos listas de manera iterativa al mismo tiempo y combina cada fila de elementos en una tupla.

La función `zip()` devuelve un objeto de la clase `zip` que contiene tuplas que aparejan los elementos que poseen el mismo índice

In [109]:
```
a = ['a','b','c','d']
z = ['0','1','2','3']

zip(a, z)
```

Out[109]:
```
<zip at 0x105d5d300>
```

In [110]:
```
zipped = []

for x in zip(a, z):
    zipped.append(x)

zipped
```

Out[110]:
```
[('a', '0'), ('b', '1'), ('c', '2'), ('d', '3')]
```

7 - Listas y diccionarios por comprensión

Las **listas por comprensión** son una de las características más apreciadas de Python. Permiten formar concisamente una nueva lista filtrando los elementos de una colección, transformando los elementos que pasan el filtro en una expresión concisa.

La sintáxis básica es la siguiente:

```
[expresion for valor in coleccion if condicion]
```

Esto es equivalente al siguente bucle for:

```
resultado = []

for valor in coleccion:
    if condicion:
        resultado.append(expresion)
```

La condición de filtro puede ser omitida, dejando sólo la expresión.

Por ejemplo, dada una lista de strings, podríamos filtrar los textos con 3 o menos caracteres y convertirlos en mayúscula de la siguiente manera:

In [111]:
```
strings = ['intro', 'a', 'data', 'science', 'con', 'Python']

[x.upper() for x in strings if len(x) > 3]
```

Out[111]:
```
['INTRO', 'DATA', 'SCIENCE', 'PYTHON']
```

Ejemplo usando enumerate y zip

Para cada elemento de las dos siguientes listas, sumar ambos elementos y dividirlos por su posición en la lista, comenzando a contar desde 1.

Por ejemplo, dadas estas dos listas:

```
lista_1 = [2, 4, 6, 8, 10]
lista_2 = [3, 6, 9, 12, 15]
```

Retornar:

```
[5.0, 5.0, 5.0, 5.0, 5.0]
```

In [112]:
```
lista_1 = [2, 4, 6, 8, 10]
lista_2 = [3, 6, 9, 12, 15]
```

In [113]:
```
[(x+y)/(i+1) for i, (x,y) in enumerate(zip(lista_1, lista_2))]
```

Out[113]:
```
[5.0, 5.0, 5.0, 5.0, 5.0]
```

Los **diccionarios por comprensión** son una extensión natural, produciendo diccionarios de una manera similar a las listas por comprensión. Un **diccionario por comprensión** se ve así:

```
dict_comp = {key_expr:valor_expr for valor in colección if condición}
```

Veamos un ejemplo donde creamos un diccionario donde cada clave es el nombre de una especie animal y su valor es la longitud de ese nombre.

In [114]:
```
keys = ['dog', 'cat', 'bird', 'horse']
dicc = {k:len(k) for k in keys}
dicc
```

Out[114]:
```
{'dog': 3, 'cat': 3, 'bird': 4, 'horse': 5}
```

Otro ejemplo, usando `zip`

In [115]:
```
column_names = ['height','weight','is_male']
values = [[62, 54, 60, 50], [180, 120, 200, 100], [True, False, True, False

dicc = {k:v for k, v in zip(column_names, values)}
dicc
```

Out[115]:
```
{'height': [62, 54, 60, 50],
 'weight': [180, 120, 200, 100],
 'is_male': [True, False, True, False]}
```

¡Muchas gracias por su atención!