

Datos, información y conocimiento

Herramientas de
ciencia de datos



tech



CONTENIDO

1. Objetivos

2. Datos, información y conocimiento

3. Tipos de datos

Tipos de datos básicos
Tipos complejos
Operaciones con datos
Estructuras de control

4. Fuentes de datos

5. Bibliografía



OBJETIVOS

- Distinguir entre los datos y la información para conocer los tipos de datos que existen reconociendo la importancia de los lenguajes en el momento de operar y crear flujos complejos.
- Analizar estructuras como los *data frames* que permiten cargar un conjunto de datos desde distintas fuentes y almacenar de una forma sencilla las características y valores de un conjunto de datos.

DATOS, INFORMACIÓN Y CONOCIMIENTO

Los datos son la base fundamental del análisis, pues son el sustento de la información del problema y el objeto a analizar mediante los lenguajes de programación.

Asimismo, es muy importante conocer cómo se representan los diferentes tipos disponibles y sus características principales que podrán diferir de unos lenguajes a otros.

Además, existe una clara distinción entre las tres formas en la que se puede encontrar la información en el mundo real o aportarle valor, de forma que, represente conceptos con un significado e implicaciones añadidas sobre las que se pueden extraer conclusiones.

Las diferencias entre los conceptos son:

- **Dato:** los datos son los hechos objetivos que se extraen de la realidad, y que sin un contexto dado no aportan una fuente objetiva sobre la que se pueda extraer conclusiones.
- **Información:** los datos se convierten en información cuando ganan este contexto y son interpretables, de forma que, se les añade valor conceptual dándole un significado que permita obtener conclusiones sobre estos.
- **Conocimiento:** hace referencia a la aportación de la experiencia a la información con la que se cuenta, de forma que, las conclusiones que se obtienen se fundamentan en hechos previos o situaciones similares. De hecho, en el análisis de datos es muy común contar con expertos que apoyen a los analistas y colaboren en el análisis.

En las siguientes secciones se verá cómo se pueden cargar los datos y con qué tipos se cuenta para este propósito, de manera que, se pueda temas posteriores, preprocesarlos y convertirlos en información sobre la que se pueda extraer conocimiento.

TIPOS DE DATOS

Como se ha comentado, es fundamental conocer los tipos de datos con los que se cuenta en un lenguaje, ya que son la base sobre la que se sustenta el flujo lógico.

TIPOS DE DATOS BÁSICOS

Los tipos de datos básicos o atómicos conforman las unidades mínimas de información que componen un lenguaje de programación. Estas unidades permiten almacenar la información y son la base sobre la que se puede operar y crear otros tipos más complejos.

Los tipos básicos que existen son comunes en la mayoría de lenguajes, difiriendo en su declaración y métodos asociados.

En la (tabla 1), se resumen los referentes a Python [1].

Una de las características sobre Python es que es un lenguaje que cumple con el paradigma orientado a objetos. Esto implica que los tipos de datos son representados como objetos. Es decir, llevan asociados una definición del dato, su valor por defecto y una serie de métodos que permiten construir, operar, eliminar de forma segura y eficaz la información almacenada en ellos.

A continuación, se abrirá el entorno en Anaconda y se lanzará el Jupyter Notebook para ver cómo se definen estos datos en Python 3. Como se verá en los ejemplos, en este lenguaje de programación no se tiene que declarar previamente de qué tipo es la variable, si no que el intérprete le asigna una clase en función del valor en cada momento.

Los comentarios cortos en Python se añaden con el símbolo '#' y son una buena práctica para añadir notas cortas al código para explicar el razonamiento seguido, de forma que, sea legible para otras personas e incluso para nosotros mismos en un futuro. Si se quieren añadir comentarios de varias líneas, se hará usando triples comillas:

```
""" Ejemplo1:

Definición de números enteros: """

a = 1

b = -2

print("Valor de a:", a) #Devuelve: "Valor de a: 1"

print("Valor de b:", b) #Devuelve: "Valor de b: -2"

print("Tipo de la variable a:", type(a)) #Devuelve: "Tipo de
la variable a: <class 'int'>"
```

La función "print" ayuda a mostrar el valor por pantalla. Al usar *notebooks*, si no se incluye esta función solo se imprimirá el último valor. Lo más recomendable es que, siempre que se quiera mostrar algo por pantalla, se haga uso de esta función.

Tipo	Descripción	Declaración
Entero	Números enteros, sin parte decimal	int Si no existe parte decimal, se asume por defecto como entero. Por ejemplo: 20.
Numérico	Números reales, también conocidos como flotantes o dobles en función de la precisión.	float La parte decimal se representa con ".". Por ejemplo: 0.023.
Complejo	Números complejos compuesto de una parte real y otra imaginaria.	complex La parte imaginaria se especifica con j. Por ejemplo: 1 + 3j.
Lógico	Subconjunto de los números enteros que expresa un valor binario.	bool Puede tomar los valores True y False.
Carácter	Datos alfanuméricos o cadenas de caracteres	str Pueden usarse comillas simples o dobles. Por ejemplo: "ciencia de datos"

Tabla 1. Tipos básicos en Python.

El caso de los números decimales es similar. Si se quiere declarar múltiples variables en una misma línea se tendrá que separarlas con el carácter ";" para separarlas:

```
c= 0.2; d=109.3245552219284 #Definición de números reales
print("El valor de c es:", c, " y el de d: ", d)
print("El valor de c es: {0:.2f}".format(c), " y el de d: {0:.2f}".format(d))
print("Tipo de la variable d:", type(d))
print("Tipo de la variable división números enteros:", type(a/b))
```

El código anterior devolverá "El valor de c es: 0.2 y el de d: 109.3245552219284" para la primera impresión y lo formateará para mostrar únicamente dos decimales con la segunda, mostrando "El valor de c es: 0.20 y el de d: 109.32" [1]. Por último, el tipo de variable que muestra es "Tipo de la variable d: <class 'float'>", que sería el mismo que la división de números enteros.

Se puede comprobar cómo el comportamiento es similar para los números complejos, donde el siguiente código muestra la salida "El valor de num es: (1+3j) y su tipo: <class 'complex'>":

```
num= 1+3j
print("El valor de num es:", num, " y su tipo: ", type(num))
print("Parte real:", num.real, " , parte imaginaria: ", num.imag)
```

También se puede acceder a su parte real e imaginaria haciendo uso de los atributos de la clase compleja, real e imaginaria, lo que mostraría para el ejemplo "Parte real: 1.0, parte imaginaria: 3.0" [1].

Continuando con los valores lógicos, se puede observar cómo estos tan solo toman los valores *true* o *false*, aunque otros tipos puedan convertirse a estos. La sentencia anterior devolverá "El valor de x es: *True* y su tipo: <class 'bool'>".

```
x = True; y = False
print("El valor de x es:", x, " y su tipo: ", type(x))
```

El último tipo que se verá son las cadenas. Estas pueden definirse con comillas simples o dobles. Si se quiere incluir dentro de una cadena un símbolo especial, como pueden ser las propias comillas, se tendrá que precederlo del símbolo '\':

```
cad1 = "Hola" ; cad2 = 'Hola'; cad3 = "\'Hola\'"
print("Valores de cadenas:", cad1, " , ", cad2, " , ", cad3)
print("Tipo de cad1: ", type(cad1))
```

Para este código se obtendrá la salida "Valores de cadenas: Hola, Hola, 'Hola' y "Tipo de cad1: <class 'str'>".

Por último, existen funciones que permiten comprobar el tipo de variable con la que se está tratando. Por ejemplo, se pueden usar *isinstance()*, que devolverá verdadero o falso. Cabe destacar que, los valores lógicos son un tipo específico de entero, como se puede observar en el cuarto ejemplo:

```
isinstance(23,int) #Devuelve True
isinstance(cad3,str) #Devuelve True
isinstance(False,bool) #Devuelve True
isinstance(True,int) #Devuelve True
isinstance(0,bool) #Devuelve False
```

TIPOS COMPLEJOS

Se consideran tipos complejos aquellos datos que no son atómicos. Los tipos de datos complejos se forman a partir de datos básicos y pueden contener o no el mismo tipo, en función de las restricciones de la estructura que construyan.

En el caso de Python corresponden a las colecciones relativas a las tuplas, las listas y los diccionarios y los conjuntos [2]. Además, también se verá como haciendo uso de librerías se pueden usar vectores y matrices.

A la hora de usar estos datos, es importante conocer qué tipos son mutables y qué tipos no. Los tipos inmutables son aquellos que, una vez que se definen o declaran, su valor no puede modificarse.

En primer lugar, las tuplas son objetos inmutables. Es decir, no se puede cambiar el valor que se le asigne cuando se defina. Se puede acceder a ellas haciendo uso de los índices, que empiezan en cero.

```
nombre_completo = ('Juan', 'Romero', 'Torres')
#Definición de una tupla

print(nombre_completo)

#Acceso con el operador [], comenzando en el índice 0

print("Su nombre completo es", nombre_completo[0], "
de primer apellido", nombre_completo[1], " y de segundo
apellido", nombre_completo[2])
```

Si se intenta reasignar el valor de alguno de sus elementos mostrará un error. Aunque, en este ejemplo, se hayan definido solo valores de tipo carácter, una tupla puede combinar diferentes tipos de datos, incluso otras tuplas.

Por otro lado, si se quiere un objeto en el que se puedan modificar sus elementos, es posible definir una lista, que es un conjunto que puede agrupar elementos de cualquier tipo.

El acceso es similar a las tuplas, haciendo uso del operador "[]", que también permite asignar un nuevo contenido haciendo uso de los índices.

```
nombre_completo = ['Juan', 'Romero', 'Torres']
#Definición de una lista

#Acceso con el operador [], comenzando en el índice 0

print("Su nombre completo es", nombre_completo[0], "
de primer apellido", nombre_completo[1], " y de segundo
apellido", nombre_completo[2])

#Modificación con el operador []

nombre_completo[0] = 'María' #Ahora el valor sería
['María', 'Romero', 'Torres']
```

Existen otros tipos complejos como los diccionarios, que son colecciones de datos sin orden, a los que no se accede a través de un índice, sino que se les asignará un identificador o clave únicos para poder referenciarlos.

Aunque también se acceden a ellos haciendo uso del operador "[]", en este caso, se puede usar el identificador que se ha asignado a cada campo. Por lo tanto, permite modificar el valor de un campo y facilita la creación de nuevos campos en el diccionario.

```
nombre_completo = {'Nombre': 'Juan', 'Apellido1':
'Romero', 'Apellido2': 'Torres'} #Definición de un
diccionario

#Acceso con el operador [], comenzando en el índice 0

print("Su nombre completo es", nombre_completo['Nombre'],
" de primer apellido", nombre_completo['Apellido1'], " y de
segundo apellido", nombre_completo['Apellido2'])

#Modificación con el operador []

nombre_completo['Nombre'] = 'María'

nombre_completo['Edad'] = 32 #Podemos añadir nuevos
campos
```

Por último, en Python existe el tipo conjunto o set, que agrupa valores no ordenados y sin elementos repetidos. Este tipo de dato permite representar valores únicos finitos de un conjunto que no tienen un orden predefinido. Por ejemplo, si se tuviese un censo de los nombres de toda la población, se podría conocer el número de nombres diferentes que existen.

```
nombres_existentes = {'Ana', 'Ana', 'Pedro', 'María', 'Juan'}
#Definición de un set

print(nombres_existentes) #Devuelve {'Juan',
'María', 'Pedro', 'Ana'} sin repetición
```

Además de los tipos complejos que se han visto, cabe destacar que, existen otros tipos en Python que se usan mucho como los vectores y las matrices.

Para poder usarlas, se tendrá que importar la librería *numpy*. Esta librería permite definir un *array*, que es una colección de elementos del mismo tipo.

```
import numpy as np #as nos permite definir un alias para
hacer uso de la librería

array = np.array([1,4,5]) #Definición del array 1,4,5

array = np.arange(start=3,stop=18,step=3) #Definición del
array 3,6,9,12,15,18

array = np.append(array,[4,6,3]) #Añadir al array nuevos
elementos

array = np.delete(array,0) #Borrar del array el primer
elemento
```

Combinando estos *arrays*, se pueden crear matrices que simplemente son vectores de más dimensiones.

```
matriz = np.array([[1,3,5],[2,4,6]]) #Definición de una matriz
print("Primer elemento de la matriz: ",matriz[0][0]) #Acceso
con el operador []

matriz [1][2] = 5 #Asignación de un nuevo valor
```

Ahora bien, se puede resumir esta sección en la (tabla 2) se comparan los tipos complejos que se han visto.

Es importante resaltar, que en Python los índices para acceder a las estructuras comienzan en 0.

Además de los tipos de datos básicos y complejos, es importante conocer en qué programación existen más estructuras. Una de las más importantes son los *data frames*. Estas estructuras representan matrices bidimensionales donde cada columna representa una característica y puede ser de un tipo de dato diferente.

En el análisis de datos se usan, pues permiten cargar de forma sencilla los datos estructurados almacenados en un archivo. Su configuración permite tratar las columnas como las variables del *dataset* y las filas serán las instancias que representen dichas características. Para poder usar esta estructura en Python, se necesitará importar la librería Pandas.

Los *data frames* se crean a partir de diccionarios, en los que se han visto como los nombres de las columnas serán las claves y los valores las filas que conforman el set de datos.

```
import pandas as pd

productos = {'Producto':['Pan','Leche','Huevos'],'Precio':
[0.65,0.75,1.19]}

df = pd.DataFrame(productos) #Definición de un data
frame

df = pd.DataFrame(productos, index=['Uno', 'Dos', 'Tres'])
#Podemos modificar el nombre de las filas

df.columns = ["A","B"] #También podemos modificar los
nombres de las columnas

print(df.Producto[0]) #Devuelve el elemento 0 de la
columna producto

print(df.iloc[0]) #Devuelve el objeto de la fila 0

df['Fecha caducidad'] = ['10/11/21','23/12/21','09/11/21']

#Podemos añadir una nueva columna, definiendo un valor
para cada fila del df['Fecha caducidad'] #Podemos eliminar
una columna

df.loc[-1] = ['Manzanas', 1.99,4] #Podemos añadir una fila
en el último lugar (índice -1)
```

Tipo	Mutable	Definición	Acceso primer elemento
Tupla	No	t = ('a', 'b', 'c')	t[0]
Lista	Sí	l = ['a', 2, 1.8]	l[0]
Diccionario	Sí	d = {'a':1, 'b': 2}	d.a
Set	Sí	s = {'a','a','b'}	No es posible acceder con índice. Hay que iterar sobre todo el set.
Vector	Sí	v = np.array([1,2,3])	v[0]
Matriz	Sí	m = np.array([[1,2],[3,4]])	m[0][0]

Tabla 2. Tipos complejos en Python.

Por último, a pesar de que se deben evitar transformar tipos de datos en otros cuando sea innecesario, ya que se puede introducir información errónea u omitir parte del dato para poder hacer la conversión, en ocasiones, es necesario e incluso se hace de forma autónoma por el propio intérprete.

Por ejemplo, ¿qué pasaría si se sumara un número entero con otro real? ¿qué ocurre si se necesita que en un determinado momento del flujo el dato cambie de tipo? Existen tres formas para cambiar los datos de manera implícita y explícita.

En primer lugar, los lenguajes permiten la coerción que ocurre, por ejemplo, cuando se suma un número entero con otro real. Este cambio lo realiza el intérprete para poder procesar una determinada orden, en este caso, añadir una parte decimal al entero para poder sumar.

Por otro lado, se conoce como *casting* a la conversión explícita que se hace de un tipo. Por ejemplo, "int (2.323)" para quedarse únicamente con la parte entera. Es deseable evitar este tipo de cambio y usar directamente el tipo de dato correcto, ya que como se puede observar en este caso, se perderá parte de la información aportada, los decimales.

Por último, se conoce como una conversión cuando el cambio se hace de forma implícita, por ejemplo, vector = c (1,2,3, true), donde true se tratará como el número entero 1.

OPERACIONES CON DATOS

Las operaciones son aquellas funciones matemáticas, lógicas o relacionales básicas que permiten transformar los datos, teniendo en cuenta que, no tienen el mismo significado en función del lugar o tipo donde se apliquen. Por ejemplo, no es lo mismo sumar enteros que cadenas.

También se prestará especial atención a aquellas operaciones que no pueden realizarse sobre algunos tipos de datos, dependiendo más de la implementación de cada lenguaje que de la operación en sí.

Por ejemplo, mientras que, en algunos lenguajes se pueden sumar cadenas, como ocurre en Python, en otros no está definido un comportamiento para dicha operación.

Python es un lenguaje bastante moldeable. En el siguiente tema se verá cómo R es más estricto en este sentido, permitiendo menos operaciones entre tipos.

En la (tabla 3), se muestra un resumen que se ejemplifica en las subsecciones. Si en una celda no se indica nada, significa que todos los tipos básicos que se vieron en el tema anterior (enteros, reales, lógicos, complejos y cadenas), tienen definido un comportamiento para dicha operación [3].

En primer lugar, se evidencian las distintas operaciones aritméticas que se muestran en la tabla. Estas son las operaciones que involucran las acciones básicas que se pueden realizar mediante el uso de las matemáticas como la adición, sustracción o multiplicación, entre otras.

En Python es posible aplicar este tipo de operaciones a todos los tipos de datos básicos: los enteros, reales, lógicos y, en algunos casos, sobre los números complejos y cadenas.

Para los tres primeros tipos, se pueden aplicar todas ellas, aunque en los valores lógicos esto es posible porque se convierten los valores verdadero y falso a 1 y 0, correspondientemente.

En los valores complejos, no se podrán aplicar únicamente los valores de módulo y división entera.

```
a,b = 9, -2
c,d = 0.12431, -23.12
e,f = True, False
g,h = 1+3j, 2+2j
print("a+b:",a+b,"c+d:",c+d,"e+f:",e+f,"g+h:",g+h)
#suma: a+b: 7 , c+d:-22.99569 , e+f: 1, g+h: (3+5j)
print("a-b:",a-b,"c-d:",c-d,"e-f:",e-f,"g-h:",g-h)
#resta: a-b: 11 y c-d: 23.24431000 , e-f: 1, g-h: (-1+1j)
print("a/b:",a/b,"c/d:",c/d,"e/e:",e/e,"g/h:",g/h) #La división
de un entero devuelve un número real: a/b: -4.5 , c/d:
-0.005376730 , e/e: 1.0, g/h: (1+0.5j)
print("a*b:",a*b,"c*d:",c*d,"e*f:", "g*h:",g*h)
#multiplicación: a*b: -18 , c*d: -2.8740472, e*f: 0, g*h:
(-4+8j)
print("a%b:", a%b,"c%d:", c%d,"e%e:",e%e) #módulo: a%b: -1 ,
c%d: -22.99569, e%e: 0
print("a**b:",a**b,"c**d:",c**d,"e**f:",e**f,"g**h:",g**h)
#potencia: a**b: 0.012345690 , c**d:8.61032894932 , e**f:
1, g**h: (0.0725150-0.819214j)
print("a//b:",a//b,"c//d:",c//d,"e//e:",e//e) #división entera:
a//b: -5, c//d: -1.0 , e//e: 1
```

En las cadenas, estos operadores aritméticos cambian el sentido, definiéndose en algunos casos con un comportamiento algo distinto. Como se evidencia en el ejemplo, en Python solo es posible aplicar los operadores aritméticos de suma y multiplicación.

```
cad1 = "Esto es "; cad2 = "un curso "
print("cad1+cad2:",cad1+cad2)
#concatenación: Esto es un curso
print("cad1*3:",cad1*3)
#repetición: Esto es Esto es Esto es
```

Por otro lado, también se pueden definir operadores lógicos de la misma forma que se ha hecho con los operadores aritméticos. Los operadores lógicos permiten evaluar si se cumple una cierta condición. Además, se podrán usarlas con todos los tipos, excepto en el caso de los operadores *bit a bit* (& y |), donde solo se podrán usarlos sobre valores enteros y lógicos.

```
a,b = 0, -2
c,d = 0.12431, -23.12
e,f = True, False
g,h = 1+3j, 2+2j
cad1 = "Esto es "; cad2 = "un curso"
print("a&b:",a&b,"e&f:",e&f)
# &: a&b: 0 , e&f: False
print("a and b:",a and b,"c and d:",c and d,"e and f:", and -f,
" g and h:",g and h,"cad1 and cad2:",cad1 and cad2 )
# and: a and b: 0 , c and d: -23.12, e and f: False, g and h:
(2+2j), cad1 and cad2: un curso
print("a|b:",a|b,"e|f:",e|f)
# | : a|b: -2 , e|f: True
print("a or b:",a or b,"c or d:",c or d,"e or f:",e or f,"g or h:",g or
h,"cad1 or cad2:",cad1 or cad2)
# or: a or b: -2 , c or d: 0.12431 , e or f: True, g or h: (1+3j),
cad1 or cad2: Esto es
```

Por último, se evidencian los operadores relacionales que permiten comparar el valor entre distintas variables. Cabe destacar que, para los números complejos solo se podrá comparar si son o no exactamente iguales.

Tipo	Operación	Símbolo	Tipos básicos que no la admiten
Aritmética	Suma	+	
	Resta	-	Las cadenas
	División	/	Las cadenas
	Multiplicación	*	
	Módulo	%	Las cadenas y los números complejos
	Potencia	**	Las cadenas
	División entera	//	Las cadenas y los números complejos
Lógico	Y lógico	and o & (bit a bit)	Cuando es <i>bit a bit</i> (&) no lo admiten reales, complejos ni cadenas.
	O lógico	or o (bit a bit)	Cuando es <i>bit a bit</i> () no lo admiten reales, complejos ni cadenas.
Relacional	Menor	<	Los números complejos
	Mayor	>	Los números complejos
	Menor o igual	<=	Los números complejos
	Mayor o igual	>=	Los números complejos
	Igual	= o is	
	Distinto	!= o is not	

Tabla 3. Resumen de los operadores en Python.

Además, en las cadenas, la definición de mayor o menor no implica cantidad, como ocurre con los números sino que hace referencia a su orden alfabético.

```
a,b = 0, -2
c,d = 0.12431, -23.12
e,f = True, False
g,h = 1+3j, 2+2j
cad1 = "Si "; cad2 = "Casino"

print("a<b:",a<b,"c<d:",c<d,"e<f:",e<f,
      "cad1<cad2:",cad1<cad2)

# menor: a<b: False, c<d: False, e<f: False, cad1<cad2:
False

print("a>b:",a>b,"c>d:",c>d,"e>f:",e>f,
      "cad1>cad2:",cad1>cad2)

# mayor: a>b: True, c>d: True, e>f: True, cad1>cad2: True

print("a<=b:",a<=b,"c<=d:",c<=d,"e<=f:",e<=f,
      "cad1<=cad2:",cad1<=cad2)

# menor o igual: a<=b: False, c<=d: False, e<=f: False,
cad1<=cad2: False

print("a>=b:",a>=b,"c>=d:",c>=d,"e>=f:",e>=f,
      "cad1>=cad2:",cad1>=cad2)

# mayor o igual: a>=b: True, c>=d: True, e>=f: True,
cad1>=cad2: True

print("a==b:",a==b,"c==d:",c==d,"e==f:",e==f,"g==h:",g==h,,
      "cad1==cad2:",cad1==cad2)

#igual: a==b:False, c==d:False, e==f:False, g==h: False,
cad1==cad2: False
```

```
print("a is b:",a is b,"c is d:",c is d,"e is f:",e is f,"g is h:",g is h,
      "cad1 is cad2:", cad1 is cad2)

#igual: a is b:False, c is d:False, e is f:False, g is h: False,
cad1 is cad2: False

print("a!=b:",a!=b,"c!=d:",c!=d,"e!=f:",e!=f,"g!=h:",g!=h,
      "cad1!=cad2:",cad1!=cad2)

#distinto: a!=b:True, c!=d:True, e!=f:True, g!=h: True,
cad1!=cad2: True

print("a is not b:",a is not b,"c is not d:",c is not d,"e is not
f:",e is not f,"g is not h:",g is not h,"cad1 is not cad2:",cad1
is not cad2)

#distinto: a is not b:True, c is not d:True, e is not f:True, g
is not h: True, cad1 is not cad2: True
```

ESTRUCTURAS DE CONTROL

Ahora que se ha visto cómo se definen los distintos operadores en Python, se podrán usar las estructuras de control que son flujos que se sustentan en comparar condiciones haciendo uso de dichos métodos entre variables. En ese sentido, estas estructuras permiten iterar y crear diferentes caminos lógicos dentro del código en función de las condiciones de ejecución.

Así pues, es importante resaltar que en Python es obligatoria la indentación. Esto hace referencia al número de tabulaciones o espacios que se dejan en el margen izquierdo.

De esta forma, el intérprete es capaz de considerar las acciones que se tienen que ejecutar dentro de una estructura, de forma que, cuando acaba la tabulación, entenderá que la acción está fuera de la estructura de control.

La tabulación es una práctica en programación que permite que el código sea más legible y claro, también permite evitar el uso de símbolos adicionales como corchetes o paréntesis para delimitar la información como ocurre con otros lenguajes.

Las estructuras de control más comunes en programación son las condicionales (*if*, *else*) y las estructuras de iteración (*for*, *while*), vamos a ver como se definen estas en concreto para Python [4].

En primer lugar, para evaluar una condición se puede usar *if-else* o *if-elif-else*.

```
color = "Rojo"

if color == "Rojo": # Evaluamos si la variable tiene
    almacenada esa cadena

    print("El color es rojo") # Como es True, imprimiré esta frase
else:

    print("El color no es rojo") # Esta expresión no se
    mostrará

color = "Azul"

if color == "Rojo": # Esta vez no entrará en la primera
    evaluación

    print("El color es rojo") # No entrará aquí
elif color == "Azul": # En este caso entra aquí

    print("El color es azul")
else:

    print("El color no es rojo ni azul") # Esta expresión no
    se mostrará
```

A continuación, se verán las estructuras de iteración. La primera de ellas es la estructura *for-in* que permite iterar sobre un conjunto de cualquier tipo: rango de enteros, lista de palabras, caracteres dentro de una cadena, etc.

```
for numero in range(0,5): # range() itera desde el primero
    hasta el penúltimo

    print("El número es:", numero) # Imprimiré 0, 1, 2, 3, 4

productos = ['pan','naranjas','aceite']

for producto in productos: # podemos iterar

    print("He comprado:", producto) # Imprimiré pan,
    naranjas, aceite
```

La otra estructura de control para poder iterar sobre un conjunto es la cláusula *while*. Sin embargo, difiere de la anterior, pues con *for* se especifica el rango o elementos a recorrer; mientras que, con *while* se espera que se cumpla o no una determinada condición.

```
a = 3

while a < 10: # iteramos hasta 9 porque cuando valga 10,
    la condición devolverá False

    print("Valor de a:", a) # Imprimiré 3, 4, 5, 6, 7, 8, 9

    a += 1 # El operador += nos permite sumar a la variable

condicion = True

contador = 0

while condicion == True: # Podríamos omitir también la
    comparación con True

    print("Contador:", contador) # Imprimiré 0, 2, 4, 6, 8, 10

    if contador == 10: # Cuando contador llegue a 10,
        cambiaremos la var condición

        condicion = False

    contador += 2
```

FUENTES DE DATOS

En esta última sección se va a ver cómo se puede extraer la información de un fichero, cargándola en memoria y permitiendo editar el contenido. De esta manera, será muy útil trabajar con los distintos *datasets* que, por lo general, se tendrá en Internet en este formato antes de procesarlos.

Para operar con ficheros en Python [5], es necesario que se preste atención al modo en el que se abre, ya que en función de este modo se podrán realizar unas acciones u otras.

Los modos disponibles son:

- **"r"**: lectura. Es el modo por defecto.
- **"w"**: escritura. Si se escribe algo borrará el contenido previo si existía el fichero, y si no existía lo creará nuevo.
- **"a"**: añadir información. Se puede escribir al final del último texto sin eliminar el contenido. Si el fichero no existe, lo crea.
- **"r+"**: lectura y escritura
- **"w+"**: lectura y escritura. Si el fichero existe, sobrescribe el contenido, y si no crea uno nuevo.
- **"a+"**: lectura y escritura. En este caso, comenzará al final del fichero, y se puede añadir contenido nuevo.

Además, es importante asegurarse de cerrar el fichero cuando se termine de tratar la información, pues así se dejará libre el espacio en la memoria.

```
texto = open('archivo.txt','r') # r es el modo read, pero
podemos usar w (write) - lo destruye si está escrito- o a
(add), añade contenido al existente, podemos leer y escribir

print("El archivo contiene: \n\n:",texto.read()) # La función
read nos permite leer el contenido completo

texto.close() # Es importante cerrar el archivo para liberar
la memoria
```

Por otro lado, existen diferentes formas de leer el texto, en este caso, solo se verán algunas. Por ejemplo, en el apartado anterior, se mostró el archivo completo, pero si se quiere leer línea a línea, considerando una línea hasta que encuentra el carácter "\n".

```
texto = open('archivo.txt','r')

print("Primera línea: \n\n:",texto.readline()) # La función
readline() nos permite leer la primera línea

print("Resto de líneas como lista: \n\n:",texto.readlines())
# La función readlines() nos permite leer las líneas
obteniéndose como lista

texto.close()
```

Asimismo, se puede almacenar el texto leído en una variable, tanto con los métodos anteriores como con "split()" que almacenará en una lista cada palabra. Es decir, si el fichero contiene "Hola, esto es un curso.", esta función devolverá ["Hola,","esto","es","un","curso."].

```
texto = open('archivo.txt','r')

var = texto.read().split() # Lee el archivo separando cada
palabra en una lista

texto.close()
```

Ahora bien, se podrán utilizar también las estructuras de control que se han visto para recorrer las líneas de un texto y realizar una determinada acción por cada línea. En consecuencia, esta acción es muy útil, pues en los *dataset*, por lo general, cada línea de un fichero corresponde a una instancia y se podrá aplicar un mismo procesamiento en cada una de ellas.

```
texto = open('archivo.txt','r')

for linea in texto: # podemos iterar con un bucle for sobre
las líneas

    print("Línea:", linea)

texto.close()
```

Por último, se podrá almacenar información en texto plano y el valor de las variables haciendo uso de la función "repr()".

```
texto = open('archivo.txt','a+') # podemos leer y añadir con a+

texto.write("Esta es una nueva línea\n") # podemos añadir
una nueva línea con write

texto.write("Valor de a = " +repr(a)+"\n")

texto.close()
```

En este caso, dado que se va a trabajar con *data frames*, se importará la librería pandas que permite leer un fichero estructurado fácilmente, cargándose en estos *frames*.

Como se puede observar, el proceso es mucho más sencillo y requiere menos líneas de código que en los casos anteriores.

```
Import pandas as pd # para leer un csv

productos = pd.read_csv('archivo.csv', sep = ";") # por defecto
el separador es ";". Lo devuelve como un data frame

productos = pd.read_csv('archivo.csv', sep = ";", names=[0,1,2]
) # podemos renombrar las columnas al cargarla

productos = pd.read_csv('archivo.csv', sep = ";", index_
col='Producto') # podemos renombrar las columnas al
cargarla

productos.to_csv('nuevo_archivo.csv') # podemos
guardarlos
```

BIBLIOGRAFÍA

- [1] "Built-in Types — Python 3.8.13 documentation", Python.org. [En línea]. Disponible en: <https://docs.python.org/3.8/library/stdtypes.html>. [Accedido: 12-abr-2022].
- [2] "Estructuras de datos — documentación de Python - 3.10.4", Python.org. [En línea]. Disponible en: <https://docs.python.org/es/3/tutorial/datastructures.html>. [Accedido: 12-abr-2022].
- [3] "Expressions — Python 3.8.13 documentation", Python.org. [En línea]. Disponible en: <https://docs.python.org/3.8/reference/expressions.html>. [Accedido: 12-abr-2022].
- [4] "Más herramientas para control de flujo — documentación de Python - 3.8.13", Python.org. [En línea]. Disponible en: <https://docs.python.org/es/3.8/tutorial/controlflow.html>. [Accedido: 12-abr-2022].
- [5] "Entrada y salida — documentación de Python - 3.8.13", Python.org. [En línea]. Disponible en: <https://docs.python.org/es/3.8/tutorial/inputoutput.html>. [Accedido: 12-abr-2022].