Análisis de datos

Introducción a Python

Transformación Digital
Agencia de Transformación Digital y Telecomunicaciones

TECNOLÓGICO NACIONAL DE MÉXICO

Introducción a Python

Python es un lenguaje de programación interpretado de alto nivel, diseñado para ser fácil de leer y sencillo de usar. Su sintaxis se asemeja al lenguaje natural, lo que ayuda a los principiantes a comprender los conceptos de programación más rápidamente. Debido a que Python se ejecuta línea por línea, permite realizar pruebas y experimentos rápidamente, sin necesidad de un paso de compilación independiente.

El lenguaje se utiliza ampliamente en campos como el desarrollo web, el análisis de datos, la inteligencia artificial y la automatización. Los desarrolladores aprecian Python porque cuenta con una gran comunidad y una amplia selección de bibliotecas externas, que ayudan a resolver muchos tipos de problemas. Su diseño fomenta un código claro y organizado, lo que lo hace adecuado, tanto para el aprendizaje como para la creación de aplicaciones profesionales complejas.

Sintaxis Python

El término *sintaxis* hace referencia a la forma en la que debemos escribir las instrucciones para que el lenguaje de programación nos entienda.

En la mayoría de lenguajes existe una sintaxis común, como por ejemplo el uso de **=** para asignar un dato a una variable, o el uso de **{}** para designar bloques de código, pero Python tiene ciertas particularidades.

El siguiente código simplemente define tres valores: `a`, `b` y `c`, realiza unas operaciones con ellos y muestra el resultado por pantalla.

```
# Definimos una variable x con una cadena
x = "El valor de (a+b)*c es"
# Podemos realizar múltiples asignaciones
a, b, c = 4, 3, 2
# Realizamos unas operaciones con a,b,c
d = (a + b) * c
# Definimos una variable booleana
imprimir = True
# Si imprimir, print()
if imprimir:
    print(x, d)
# Salida: El valor de (a+b)*c es 14
```

Indentación y bloques de código

En Python, los bloques de código se representan con indentación, es decir, el espacio en blanco al inicio de cada línea de código. Aunque existe un poco de debate, en relación con el uso de tabulador o de espacios, la norma general es usar **cuatro espacios**.

En el siguiente código tenemos un condicional *if*. Justo después tenemos un `print`() indentado con cuatro espacios. Por lo tanto, todo lo que tenga esa indentación pertenece al bloque del *if.*

```
if True:
    print("True")
```

Función print()

En cualquier lenguaje de programación es importante saber lo que va pasando, a medida que se ejecutan las diferentes instrucciones. Por ello, es importante hacer uso de *print()* en diferentes secciones del código, ya que nos permiten ver el valor de las variables, así como diferente información útil.

Se puede usar `print()` para imprimir por pantalla el texto que queramos.

```
print("Esto es el contenido a imprimir")
```

También es posible imprimir el contenido de una variable.

```
x = 10
print(x)
```

Y separando por comas los valores, es posible imprimir el texto y el contenido de las variables.

```
x = 10
y = 20
print("Los valores x, y son:", x, y)
# Salida: Los valores x, y son: 10 20
```

2.1 Tipos de datos

Python tiene varios tipos básicos de datos que ayudan al lenguaje a representar información.

A continuación se ofrece una explicación:

Tipo de datos	Descripción
int	Números enteros sin decimales, por ejemplo, 5, -2, 100, etc.
float	Números decimales, por ejemplo, 3.14, -0.5.
str	Cadena de caracteres, por ejemplo, «Hola mundo».

bool	Valores lógicos: verdadero o falso.
list	Colección ordenada y mutable de elementos.
tuple	Colección ordenada e inmutable de elementos.
dict	Pares clave-valor utilizados para mapear información.
set	Colección desordenada de elementos únicos.

2.2. Operadores

Los operadores son símbolos o palabras especiales, que permiten realizar operaciones sobre valores o variables. Son herramientas básicas para realizar asignación de valores, operaciones aritméticas, comparaciones relacionales y lógicas. A continuación, se presentan diferentes tipos de operadores.

- **Operadores de asignación**

Los operadores de asignación nos permiten realizar una operación y almacenar su resultado en la variable inicial.

Operador	Ejemplo	Equivalente
=	x=7	x=7
+=	x+=2	x=x+2 = 7
-=	x-=2	x=x-2 = 5
*=	x*=2	x=x*2 = 14
/=	x/=2	x=x/2 = 3.5
%=	x%=2	x=x%2 = 1

- **Operadores aritméticos**

Los operadores aritméticos son los más comunes que nos podemos encontrar, y nos permiten realizar operaciones aritméticas sencillas, como pueden ser la suma, resta o exponente.

Operador	Nombre	Ejemplo
+	Suma	x + y = 13
-	Resta	x - y = 7
*	Multiplicación	x * y = 30
/	División	x/y = 3.333
%	Módulo	x%y = 1
**	Exponente	x ** y = 1000
//	Cociente	3

- **Operadores relacionales**

Los operadores relacionales nos permiten saber la relación existente entre dos variables. Se usan para saber si, por ejemplo, un número es mayor o menor que otro. Dado que estos operadores indican si se cumple o no una operación, el valor que devuelven es True o False.

Operador	Nombre	Ejemplo
==	Igual	x == y = False
!=	Distinto	x != y = True

>	Mayor	x > y = False
&lt	Menor	x &lt y = True
>=	Mayor o igual	x >= y = False
&lt=	Menor o igual	x &lt y = True

- **Operadores lógicos**

Los operadores lógicos nos permiten trabajar con valores de tipo booleano. Un valor booleano o bool es un tipo que solo puede tomar valores *True* o *False* (verdadero o falso). Por lo tanto, estos operadores nos permiten realizar diferentes operaciones con estos tipos, y su resultado será otro booleano. Por ejemplo, True and True usa el operador and, y su resultado será True.

Operador	Nombre	Ejemplo
and	Devuelve True si ambos elementos son True	True and True = True
or	Devuelve True si al menos un elemento es True	True or False = True
not	Devuelve el contrario, True si es Falso y viceversa	not True = False

2.3. Estructuras de control

Una estructura de control en Python, es un conjunto de instrucciones que permite tomar decisiones o repetir acciones dentro de un programa. Gracias a ellas, el código puede cambiar su flujo normal de ejecución, en lugar de ejecutarse línea por línea, de forma secuencial.

Condicionales

Los condicionales son estructuras que permiten que un programa tome decisiones y ejecute ciertas instrucciones, siempre y cuando se cumpla una condición específica. Se usan para controlar el flujo del programa, según se cumplan o no ciertas condiciones lógicas.

- **Uso del if**

La estructura *if* básica en Python, evalúa una condición y ejecuta un bloque de código si la condición resulta verdadera.

Un ejemplo sería cuando tenemos dos valores a y b, que queremos dividir. Antes de entrar en el bloque de código que divide a/b, sería importante verificar que b es distinto de cero, ya que la división por cero no está definida. Es aquí donde entran los condicionales if.

```
a = 4
b = 2
if b != 0:
    print(a/b)
```

En este ejemplo podemos ver cómo se puede usar un *if* en Python. Con el operador **!=** se comprueba que el número b sea distinto de cero y, si lo es, se ejecuta el código que está indentado. Por lo tanto un *if* tiene dos partes:

- La condición que se tiene que cumplir para que el bloque de código se ejecute, en nuestro caso b!=0.
- El bloque de código que se ejecutará, si se cumple la condición anterior.

La sentencia *if* debe terminar con dos puntos (**:**), y el bloque de código a ejecutar debe estar indentado. El bloque de código puede también contener más de una línea, es decir, puede contener más de una instrucción.

```
if b != 0:
    c = a/b
    d = c + 1
    print(d)
```

Todo lo que vaya después del *if* y esté indentado, será parte del bloque de código que se ejecutará, si la condición se cumple. Por lo tanto, el segundo print() "Fuera if" será ejecutado siempre, ya que está fuera del bloque if.

```
if b != 0:
    c = a/b
    print("Dentro if")
print("Fuera if")
```

También se pueden combinar varias condiciones entre el *if* y los **:**. Por ejemplo, se puede requerir que un número sea mayor que 5 y además, menor que 15. Tenemos en realidad tres operadores usados conjuntamente, que serán evaluados por separado, hasta devolver el resultado final, que será *True* si la condición se cumple o *False* en el caso contrario.

```
a = 10
if a > 5 and a < 15:
    print("Mayor que 5 y menos que 15")
```

Es muy importante tener en cuenta que, a diferencia de otros lenguajes, en Python no puede haber un bloque *if* vacío. El siguiente código daría un `SyntaxError`.

```
if a > 5:

```

- **Uso de *else* y *elif***

Es posible que no solo queramos hacer algo si una determinada condición se cumple, sino que además, consideremos hacer algo en el sentido contrario. Es aquí donde entra la cláusula *else*. La parte del *if* se comporta de la manera que ya hemos explicado, con la diferencia que si esa condición no se cumple, se ejecutará el código presente dentro del *else*. Ambos bloques de código son excluyentes, se entra en uno o en otro, pero nunca se ejecutarán los dos.

```
x = 5
if x == 5:
    print("Es 5")
else:
    print("No es 5")
```

Hasta ahora hemos visto cómo ejecutar un bloque de código si se cumple una instrucción, pero, en muchos casos, podemos tener varias condiciones diferentes y para cada una queremos un código distinto. Es aquí donde entra en juego el *elif*.

```
x = 5
if x == 5:
    print("Es 5")
elif x == 6:
    print("Es 6")
elif x == 7:
    print("Es 7")
```

Con la cláusula *elif* podemos ejecutar tantos bloques de código distintos como queramos, según la condición. Traducido al lenguaje natural, sería como decir: si es igual a 5 haz esto, si es igual a 6 haz lo otro, si es igual a 7 haz esto otro.

Se puede también usar todo de manera conjunta; el *if* con el *elif* y un *else* al final, pero de las instrucciones *if* y *else* solamente puede haber una, mientras que *elif* puede repetirse varias veces.

```
x = 5
if x == 5:
    print("Es 5")
elif x == 6:
    print("Es 6")
elif x == 7:
    print("Es 7")
else:
    print("Es otro")
```

Bucles for y while

En Python, *for* y *while* son comandos especiales, considerados como bucles que permiten ejecutar bloques de código repetidamente o, dicho de otra manera, son comandos que permiten ejecutar un bloque de código varias veces a sí mismo. Aunque ambos comandos son para la misma funcionalidad, en realidad cada uno cuenta con diferencias importantes, debido a que funcionan en contextos diferentes.

- **Bucle for**

El comando *for* se usa para ejecutar un bloque de código sobre una secuencia, como una lista, una cadena, un rango, etc. Esto es posible porque el bucle *for* define las veces que se ejecutará el código, este bucle se caracteriza por tener **definido el número** de veces a ejecutarse, a diferencia del *while*, que tiene número indefinido de ejecución. A continuación se presenta un ejemplo del uso del *for*:

```
for i in range(0, 5):
    print(i)

# Salida:
# 0
# 1
# 2
# 3
# 4
```

- **Bucle while**

Este comando nos permite ejecutar una sección del código repetidas veces. El código se ejecutará mientras una condición determinada se cumpla. Cuando se deje de cumplir, se saldrá del bucle y se continuará la ejecución normal. Este tipo de bucle se caracteriza por contar con un número de ejecuciones de bloques de código **no definidos**. A continuación se presenta un ejemplo del uso del *while*:

```
x = 5
while x > 0:
    x -=1
    print(x)

# Salida: 4,3,2,1,0
```

Iterables e iteradores

Los *iterables* son aquellos objetos que puedan ser indexados o accedidos por un índice. Si piensas en un array (o una lista en Python), podemos indexarlo con *lista[1]* por lo que sería un iterable. Algunos ejemplos de iterables en Python son las listas, tuplas, cadenas o diccionarios.

Los *iteradores* son objetos que hacen referencia a un elemento, y que tienen un método *next* que permite hacer referencia al siguiente.

Con la función *isinstance()* podemos saber si una variable es iterable. *True* significa que es iterable y *False* que no lo es.

```
from collections import Iterable
lista = [1, 2, 3, 4]
cadena = "Python"
numero = 10
print(isinstance(lista, Iterable))  #True
print(isinstance(cadena, Iterable)) #True
print(isinstance(numero, Iterable)) #False
```

Para entender los iteradores, es importante conocer la función *iter()* en Python. Dicha función puede ser llamada sobre un objeto que sea iterable, y nos devolverá un iterador, como se ve en el siguiente ejemplo:

```
lista = [5, 6, 3, 2]
it = iter(lista)
print(it)       #<list_iterator object at 0x106243828>
print(type(it)) #<class 'list_iterator'>
```

For anidados

Es posible anidar los *for*, es decir, meter uno dentro de otro. Esto puede ser muy útil si queremos iterar algún objeto que en cada elemento tiene, a su vez, otra clase iterable. Podemos tener por ejemplo, una lista de listas, una especie de matriz.

```
lista = [[56, 34, 1],
         [12, 4, 5],
         [9, 4, 3]]
```

Si iteramos usando sólo un *for*, estaremos accediendo a la segunda lista, pero no a los elementos individuales.

```
for i in lista:
    print(i)
#[56, 34, 1]
#[12, 4, 5]
#[9, 4, 3]
```

Si queremos acceder a cada elemento individualmente, podemos anidar dos *for*. Uno de ellos se encargará de iterar las columnas y el otro las filas.

```
for i in lista:
    for j in i:
        print(j)
# Salida: 56,34,1,12,4,5,9,4,3
```

Else y while

Algo muy común en Python es el uso de la cláusula ***else*** al final del ***while***. La sección de código que se encuentra dentro del ***else***, se ejecutará cuando el bucle termine, pero solo si lo hace “por razones naturales”. Es decir, si el

bucle termina es porque la condición se deja de cumplir, y no porque se ha hecho uso del ***break***.

```
x = 5
while x > 0:
    x -=1
    print(x) #4,3,2,1,0
else:
    print("El bucle ha finalizado")
```

While anidados

Ya hemos visto que los bucles ***while*** tienen una condición a evaluar, y un bloque de código a ejecutar. Hemos visto ejemplos donde el bloque de código está integrado por operaciones sencillas, como la resta "-", pero es posible subir de nivel y meter otro bucle ***while*** dentro del primero. Es algo que resulta especialmente útil si, por ejemplo, queremos generar permutaciones de números, es decir, si queremos generar todas las combinaciones posibles. Imaginemos que queremos generar todas las combinaciones de dos números hasta 2. Es decir, 0-0, 0-1, 0-2,... hasta 2-2.

```
# Permutación a generar
i = 0
j = 0
while i < 3:
    while j < 3:
        print(i,j)
        j += 1
    i += 1
    j = 0
```

2.4 Funciones en Python

Una función es un grupo de instrucciones que constituyen una unidad lógica del programa y resuelven un problema muy concreto. Las funciones tienen un doble objetivo:

- Dividir y organizar el código en partes más sencillas.
- Encapsular el código que se repite a lo largo de un programa, para ser reutilizado.

Existen dos tipos de funciones, según su origen:

- **Funciones nativas**

El intérprete de Python ya cuenta con una serie de funciones y tipos incluidos en él, que están siempre disponibles.

- **Funciones personalizadas**

También es posible definir nuestras propias funciones. Para ello, se utiliza la palabra clave *def*, que indica al lenguaje que se va a crear una función con un nombre y un comportamiento específicos. Esto permite agrupar un conjunto de instrucciones bajo un mismo bloque, facilitando su reutilización en diferentes momentos del programa.

```
def nombre_funcion(argumentos):
    código
    return retorno
```

Cualquier función tendrá un nombre, unos argumentos de entrada, un código a ejecutar y unos parámetros de salida. Al igual que las funciones matemáticas, en programación nos permiten realizar diferentes operaciones

con la entrada, para entregar una determinada salida, que dependerá del código que vayamos a escribir dentro.

Algo que diferencia, en cierto modo, a las funciones en el mundo de la programación, es que no solo realizan una operación con sus entradas, sino que también parten de los siguientes principios:

- El principio de **reusabilidad**, el cual nos dice que si, por ejemplo, tenemos un fragmento de código usado en muchos sitios, la mejor solución sería pasarlo a una función. Esto nos evitaría tener código repetido, y que modificarlo sea más fácil, ya que bastaría con cambiar esa función una sola vez.
- El principio de **modularidad**, el cual defiende que, en vez de escribir largos trozos de código, es mejor crear módulos o funciones que agrupen ciertos fragmentos de código en funcionalidades específicas, haciendo que el código resultante sea más fácil de leer.

Tipos de argumentos en funciones personalizadas

A continuación, se presentan los diferentes tipos de argumentos en funciones personalizadas.

Tipo de paso de argumentos	Ejemplo en Python

Pasando argumentos de entrada	`#Función`
	`def di_hola(nombre):`
	`  print("Hola", nombre)`
	`#Paso de argumentos de entrada`
	`di_hola("Juan")`

Argumentos por posición	`#Función`
	`def resta(a, b):`
	`  return a-b`
	`#Paso de argumentos por posición`
	`resta(5, 3) # 2`

Argumentos por nombre	`#Función`
	`def resta(a, b):`
	`  return a-b`
	`#Paso de argumentos por nombre`
	`resta(a=3, b=5)`

Argumentos por defecto	`#Función`
	`def suma(a=3, b=5, c=0):`
	`  return a+b+c`
	`#Paso de argumentos por defecto`
	`suma() # 8`

Argumentos de longitud variable	`#Función`
	`def suma(numeros):`
	`  total = 0`
	`  for n in numeros:`
	`  total += n`
	`  return total`
	`#Paso de argumentos de longitud variable`
	`suma([1,3,5,4])`

Sentencia return

El uso de la sentencia ***return*** permite realizar dos cosas:

- Salir de la función y transferir la ejecución de vuelta a donde se realizó la llamada.
- Devolver uno o varios parámetros, fruto de la ejecución de la función.

A continuación se presenta el ejemplo del uso de un return.

```
def di_hola():
    return "Hola"
di_hola()
# 'Hola'
```

Elaboró contenido: Dr. Humberto Marín Vega

Referencias:

- Downey, A., Elkner, J., & Meyers, C. (2009). *Introducción a la programación con Python*. Traducción de A. Becerra Sandoval. Pontificia Universidad Javeriana.
- Bahit, E. (2018). *Introducción al Lenguaje Python* [PDF]. Recuperado de https://www.researchgate.net/profile/Eugenia-Bahit/publication/333965199_Introduccion_al_Lenguaje_Python/links/5d0efe0c458515c11cf0eb75/Introduccion-al-Lenguaje-Python.pdf
- Trejos Buriticá, O. I., & Muñoz Guerrero, L. E. (2021). *Introducción a la programación con Python*. Ra-Ma Editorial.