Entrada y salida de datos

March 29, 2022

1 Introducción a Python

1.1 Python

Python es un lenguaje de programación creado por Guido van Rossum a principios de los años 90, cuyo nombre está inspirado en el grupo de cómicos ingleses "Monty Python". Es un lenguaje con una sintaxis muy limpia y que favorece un código legible. Se trata de un lenguaje interpretado o de script, con tipado dinámico, fuertemente tipado, multiplataforma y orientado a objetos. Decimos que es interpretado ya que Python ejecuta línea por línea el código, como si fuera un guion (de aquí la palabra script, que es su contraparte inglesa) De este modo, el programa no conoce las instrucciones que deberá ejecutar más adelante, por lo cual, algunos errores solo se presentarán al llegar hasta la línea que lo detona. Antes de ver algunas de las características que diferencian a Python de los otros lenguajes de programación, veamos qué es una variable y qué tipo de datos podemos almacenar en ellas.

1.2 Comentarios

Siempre que deseemos que Python ignore una línea, de código, basta con poner un **#** al inicio de la misma, esto es útil para documentar el funcionamiento de nuestro código, esto es, añadir al inicio del script, dentro de un comentario, una descripción clara y concisa del funcionamiento del código consignado debajo del comentario, de este modo, al abrirlo en un futuro lejano podremos entender nuevamente el propósito del programa.

```
instruccion
# Comentario
```

1.3 Variables y constantes

Una variable es un espacio dentro de la memoria que reservamos para almacenar datos de distintos tipos y le asignamos un nombre para poder acceder a él. Es útil pensar en las variables como un contenedor que almacena datos que pueden ser modificados posteriormente en el programa. En la imágen anterior podemos ver una buena analogía del comportamiento de las variables en Python, podemos ver estas como vasijas que almacenan datos y sólo se alterará su contenido si realizamos operaciones sobre estas.

Por ejemplo, asignemos el valor 555 a la variable a:

[ ]:
```
a = 555
```

Como vemos, en Python el proceso de asignación de variables es tan simple como asignar un nombre a la variable y luego escribir un = seguido del valor. Es un similar a las matemáticas, donde escribíamos simplemente

$$a = 555$$

Por otra parte, si deseamos observar los valores almacenados en la variable, basta con usar el comando `print` del lenguaje, el cual se usa de la siguiente manera:

[ ]:
```
print(a)
```

```
555
```

Al ejecutar el bloque de código, es posible observar cómo Python, justo debajo de dicho bloque, nos entrega la salida por "consola" que obtendríamos al ejecutar el código desde el computador.

`print` se debe usar siempre seguido del valor a imprimir rodeado de paréntesis, si queremos imprimir varios valores, basta con separar estos con comas y si queremos imprimir textos, estos deben estar entre comillas, sean sencillas o dobles (De estos hablaremos a continuación).

[ ]:
```
print(a,"esto es una cadena de texto", 1234, 'esto también es una cadena de␣
↪texto ')
```

```
555 esto es una cadena de texto 1234 esto también es una cadena de texto
```

1.4 Convención para nombrar variables.

Python utiliza unas convenciones de estilo pensadas en garantizar la legibilidad del código, estás reglas en encuentran en el documento llamado PEP 8 (Python Enhancement Proposal) creado en el 2001 y con su última modificación en el 2013.

Aquí se recomienda nombrar las variables usando "snake case", este término se refiere al estilo de escritura en el que cada espacio se sustituye por un carácter de subrayado (_) y la primera letra de cada palabra se escribe en minúscula. Veamos algunos ejemplos de definición de variables bajo esta convención.

```
pi = 3.1415
promedio_acumulado = 55.845
iva_producto = 0.19
```

Las variables debem describir claramente el valor que contendrán, sin embargo, tampoco es recomendable usar demasiadas palabras, pues cada una hará más difícil la lectura del código.

También es recomendable dejar un espacio antes y después de cada operador, sea un = o cualquiera de los operadores que veremos en la próxima sección, esto nuevamente en pro de la legibilidad del código.

Ahora bien, ¿con qué tipos de valores podemos trabajar en Python?

1.5 Numéricos

Python incluye tres tipos numéricos para representar números: enteros, flotantes y complejos.

1.5.1 Enteros

Los enteros son números tales como el cero, positivos o negativos sin parte decimal y con precisión ilimitada, por ejemplo, 0, 100, -10. Esta precisión es el número de bits usados para representar un valor, de este modo, precisión ilimitada quiere decir que, a diferencia de otros lenguajes, podemos representar cualquier número sin preocuparnos por el espacio de memoria que ocuparía en la RAM. Los siguientes son enteros válidos en Python.

```
>>> 0
>>> 100
>>> -10
>>> 1234567890
>>> y=500000000000000000000000000000000000000000000000000000000000
```

Podemos usar el comando `type` para verificar cúal es el tipo de estos valores:

[ ]:
```
print("tipo de 0: ", type(0))
y=500000000000000000000000000000000000000000000000000000000000
print("tipo de y: ", type(y))
```

```
tipo de 0:  <class 'int'>
tipo de y:  <class 'int'>
```

La salida del comando `type` muestra la palabra reservada `class`, esta hace parte de un paradigma del cual hablaremos en el ciclo 2, la programación orientada a objetos. En pocas palabras, una clase es una plantilla para la creación de objetos de datos según un modelo predefinido. Las clases se utilizan para representar entidades o conceptos. Cada clase es un modelo que define un conjunto de variables (el estado), y métodos apropiados para operar con dichos datos (el comportamiento). En este caso, la entidad es el tipo de dato entero: `int`. Vale la pena resaltar que en Python, no está permitido escribir enteros con un cero a la izquierda precediendo el valor:

[ ]:
```
012345
```

```
  File "<ipython-input-5-56565d45eb32>", line 1
    012345
         ^
SyntaxError: leading zeros in decimal integer literals are not permitted; use a
 ↪0o prefix for octal integers
```

1.5.2 Decimales: "punto flotante"

Se denomina punto flotante `float` al método de representación de números reales que permite que la posición del punto se mueva (flote) a cualquier posición del número, permitiendo por ello un rango mayor de los números que es posible representar con cantidad fija de dígitos. Los números decimales en computación se escriben con un punto para separar la parte entera y la fracción, sin embargo, también es posible usar `e` para usar notación científica, es decir para escribir números de la forma 1×10^n por ejemplo, el número 3.4556789e2 equivale al número 3.4557×10^2 o sea`345.57`. así, los siguientes son ejemplos de números válidos de tipo flotante:

```
>>> f=1.2
>>> 1e3
>>> 3.4556789e2
>>> x = -123.587
```

si usamos el comando type, obtendremos la clase `float`:

[ ]:

[ ]:
```
print('tipo flotante: ', type(3.4556789))
```

```
tipo flotante:  <class 'float'>
```

1.5.3 Números complejos

Si bien en computación no es muy común trabajar con números imaginarios (es decir, números de la forma $z = a + ib$ donde $i = \sqrt{-1}$, a es la parte real y b la imaginaria), Python de forma nativa ofrece soporte para operar con ellos, aquí, la parte imaginaria i se denota con la letra j, esto debido al estándar propuesto por la IEEE para trabajar con números complejos.

[ ]:
```
a = 5 + 2j
print(a, "tipo de a: ", type(a))
```

```
(5+2j) tipo de a:  <class 'complex'>
```

No se pueden escribir con otras letras o dejar la parte imaginaria j vacía, de lo contrario se obtendrán errores de sintaxis:

[ ]:
```
a = 5 + 2k
```

```
  File "<ipython-input-8-6b857f163a1c>", line 1
    a=5+2k
       ^
SyntaxError: invalid syntax
```

[ ]:
```
z = 1 + j
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-9-53ad48aa7f6c> in <module>
----> 1 z=1+j

NameError: name 'j' is not defined
```

En este caso, Python asume `j` como una variable cualquiera, pero al no estar definida, se marca el error que se observa arriba. Por lo tanto, la forma correcta para definir el número complejo en python no es `z = 1 + j` si no `z = 1 + 1j`

1.6 Strings

En Python, un string o cadena de caracteres es una secuencia de datos inmutable, esto quiere decir que una vez definido un string, no es posible modificar su contenido interno. Como observamos anteriormente, un string es una secuencia de caracteres envueltos dentro de comillas simples, dobles o triples.

```
>>>'Esto es un string en Python' # string en comillas sencillas
>>>"Esto es un string en Python" # string en comillas dobles
>>>'''Esto es un string en Python''' # string en comillas triples
>>>"""Esto es un string en Python""" # string en comillas dobles-triples (es equivalente al an
```

[ ]:
```
s = 'Hola mundo'
print(s, 'tipo de s:', type(s))
```

```
Hola mundo tipo de s: <class 'str'>
```

Los strings de tres comillas permiten construir textos con saltos de línea:

[ ]:
```
s = '''Esto es
un string
con saltos de línea
'''
print(s)
```

```
Esto es
un string
con saltos de línea
```

Cabe aclarar que también es posible crear textos con saltos de línea añadiendo `\n` dentro del mismo renglón, el cual es el caracter especial reservado para crear saltos de línea en un computador, por lo tanto, queda a discreción de cada programador elegir la alternativa más cómoda para sus necesidades en cada programa.

Es posible conocer la cantidad de caracteres de un string al usar el operador `len`, este cuenta también los espacios y caracteres especiales que estén presentes dentro del string:

[ ]:
```
a = 'esto es un string con un salto de linea \nEn su interior'
print(a, 'tamaño:', len(a))
```

```
esto es un string con un salto de linea
En su interior tamaño: 55
```

Si queremos acceder al caracter presente en una posición específica del string, usamos corchetes y en su interior debemos ingresar la posición que queremos extraer, el primer caracter de cualquier string ocupa la posición 0, de este modo, si queremos capturar la cuarta posición, debemos poner el índice 3:

[ ]:
```
a = 'parangaricutirimicuaro'
print('posición 0:', a[0], 'cuarta posición:', a[3])
```

```
posición 0: p cuarta posición: a
```

Si escribimos `a[4]` en este caso obtendríamos una n, que equivale a la quinta posición:

[ ]:
```
print(a[4])
```

```
n
```

Observemos por qué los string son inmutables. Si quisiéramos cambiar la cuarta posición del string `a` debería bastar con hacer lo siguiente

```
a[3] = 'X'
```

Sin embargo, si ejecutamos dicho código, obtenemos lo siguiente:

[ ]:
```
a[3] = 'X'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-15-cac56f5b4a7e> in <module>
----> 1 a[3] = 'X'

TypeError: 'str' object does not support item assignment
```

1.7 Booleanos

Este tipo de dato representan los estados lógicos falso y verdadero, es muy importante ya que, gracias a este, podremos más adelante hacer operaciones cíclicas o realizar operaciones alternativas según una condición establecida.

```
>>> True
>>> False
```

Como vemos, ambos valores usan su palabra en inglés y con el primer caracter en mayúscula.

[ ]:
```
print(type(True))
```

```
<class 'bool'>
```

2 Captura de datos

Es común en la programación solicitarle al usuario que ingrese valores con los cuales se pueda realizar alguna operación, este proceso se puede realizar con la función `input`:

```
variable = input('mensaje')
```

El mensaje que va en el interior de `input` aparecerá antes del puntero a la espera del valor a almacenar y se usa para explicarle al usuario qué debe ingresarle al programa, núnca se puede

asumir que el usuario tiene pleno conocimiento del funcionamiento del programa que estamos codificando. Ejecuta el siguiente bloque de código e ingresa un valor numérico cualquiera.

[ ]:
```
variable = input('digite un numero: ')
print(variable)
```

```
digite un numero:  5
5
```

Ahora verifiquemos qué tipo de variable nos entregó `input`

[ ]:
```
print(type(variable))
```

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

A pesar de que ingresamos un número `input` lo almacenó como un string, esto se debe a que desde Python 3, todas las entradas por consola se almacenan como strings, si deseamos convertir la variable a otro tipo, debemos usar el comando `eval`, el cual determinará el tipo más apropiado para la variable o convertirla explícitamente, aplicando un ***casting*** (este termino se refiere a la transformación de un tipo de dato en otro) con los siguientes comandos: * `int()` : convierte la variable en tipo entero * `float()` : convierte la variable en tipo punto flotante * `str()` : convierte la variable a string (no es necesaria en un input) * `bool()` : convierte la variable a booleano * `complex()` : convierte la variable en número complejo

[ ]:
```
variable = eval(input('digite un número: '))
print(variable, type(variable))
```

```
digite un número: 5
5 <class 'int'>
```

[ ]:
```
variable = eval(input('digite un número: '))
print(variable, type(variable))
```

```
digite un número: 3.14
3.14 <class 'float'>
```

[ ]:
```
variable = eval(input('digite un número: '))
print(variable, type(variable))
```

```
digite un número: 5+4j
(5+4j) <class 'complex'>
```

Si usamos el casting de entero en una variable flotante, eliminaremos la parte fraccionaria, el lenguaje **NO** redondea al realizar este proceso.

[ ]:
```
print(float(5))
```

```
5.0
```

[ ]:
```
print(int(3.9999))
```

```
3
```

3 Impresión de datos

Ya vimos que el comando para imprimir a consola cualquier tipo de información es `print`, sin embargo, vamos a analizar algunos tipos de impresión distintos que se pueden realizar a partir de este comando.

[ ]:
```
print('primer mensaje')
print('segundo mensaje')
print('tercer mensaje')
```

```
primer mensaje
segundo mensaje
tercer mensaje
```

Como vemos, Python por defecto separa las impresiones con un salto de línea, sin embargo, podemos cambiar este comportamiento añadiendo la instrucción `end` luego de una coma, dentro de los paréntesis de la función.

[ ]:
```
print('primer mensaje',  end=" ")
print('segundo mensaje', end=" ")
print('tercer mensaje')
```

```
primer mensaje segundo mensaje tercer mensaje
```

Como vimos anteriormente, si añadimos dentro de un `print` varios objetos, este los separa con un espacio, si queremos modificar esto, debemos añadir la instrucción `sep` dentro de la función, del mismo modo que hicimos con `end`. Podemos poner cualquier string que deseemos en el interior.

[ ]:
```
print('primer mensaje', 'segundo mensaje', 'tercer mensaje')
```

```
primer mensaje segundo mensaje tercer mensaje
```

[ ]:
```
print('primer mensaje', 'segundo mensaje', 'tercer mensaje', sep='\n')
```

```
primer mensaje
segundo mensaje
tercer mensaje
```

[ ]:
```
print('primer mensaje', 'segundo mensaje', 'tercer mensaje', sep='->')
```

```
primer mensaje->segundo mensaje->tercer mensaje
```

3.1 Impresión con formato

La impresión con formato es una herramienta bastante poderosa, puesto que nos permite garantizar que las salidas de nuestro programa se comporten siempre de la misma manera independientemente de los datos que le entreguemos al mismo. Hay dos formas de añadir formato a una impresión:

3.1.1 Formateo de la salida utilizando el operador de módulo(%) :

```
[ ]: print("Total de estudiantes: %3.1d, niños : %2.2d" % (240, 120))
```

```
Total de estudiantes: 240, niños : 120
```

El operador % también puede utilizarse para formatear strings. Para ello, la clase string sobrecarga el operador módulo % para realizar el formateo de cadenas. Por lo tanto, a menudo se le llama operador de módulo de string.

El operador de módulo de cadena (%) todavía está disponible en Python(3.x) y es ampliamente utilizado, ya que permite restringir la cantidad de números a imprimir en variables.

Como vimos en el ejemplo, los % acompañan números y letras, estos flag funcionan como "placeholders" en los cuales se insertará el valor, estos flag utilizan la siguiente sintaxis, tanto el ancho como precición son opcionales, pero el indicador de tipo **Sí** es necesario, ya que Python convierte el valor para que se ajuste al tipo deseado :

```
%[<ancho>][.<presición>]<type>
```

En este enlace se explica a detalle cómo funcionan estos flag. De forma resumida, el primer número indica cantidad de espacios que se reservarán en la salida para la impresión, si se tiene un número o string con un tamaño superior a este ancho, el valor se imprimirá completo de todas formas, por lo cual es importante tener presente cual es el mayor valor a imprimir en el programa antes de crear el formato, de lo contrario, las impresiones se pueden desordenar, si fuesen inferiores, se dejará el espacio restante a la izquierda:

```
[ ]: print("mayor:%3.1d, menor:%5.2d" % (123456789, 120))
```

```
mayor:123456789, menor:  120
```

El número después del punto, corresponde a la precición requerida para los número decimales, es decir, la cantidad de dígitos que se permitirá imprimir de la parte decimal, si el número tiene menos dígitos que la precición, se rellenará con ceros a la derecha, en el caso contrario, el número se redondeará a la cantidad de cifras permitidas:

```
[ ]: print("mayor:%3.2f, menor:%5.5f" % (123.546872, 120.123))
```

```
mayor:123.55, menor:120.12300
```

En el anterior ejemplo, se cambió la `d` por una `f` esto se debe a que esta letra indica el tipo de dato que se va a insertar, a continuación se presentará una tabla con los tipos más comunes.

type	Tipo a insertar
d	Entero decimal
x, X	Entero hexadecimal
o	Entero octal
f, F	Punto flotante
e, E	Notación científica
g, G	Flotante o notación científica
c	Un solo caracter

type	Tipo a insertar
s	String

El flag g usará el tipo que mejor se ajuste al ancho definido entre el flotante y la notación científica

```
[ ]: print("Número decimal: %d, número hexadecimal: %x, notación científica: %4.2e, ␣
     ↪flag g: %4.2g"%(4869,4869,123654.54236,4864.54))
```

```
Número decimal: 4869, número hexadecimal: 1305, notación científica: 1.24e+05,
flag g: 4.9e+03
```

3.1.2 Interpolación de strings en Python :

La interpolación de cadenas es un proceso que sustituye los valores de las variables en los marcadores de posición de una cadena. Por ejemplo, si tenemos una plantilla para saludar a una persona como `"Hola {nombre_persona}, ¡encantado de conocerte!"`, nos gustaría sustituir el marcador de posición de `nombre_persona` por un nombre real. Este proceso se llama interpolación de strings.

f-strings Python 3.6 añadió un nuevo método de interpolación de strings, llamado interpolación literal de strings e introdujo un nuevo prefijo literal f. Esta nueva forma de formatear strings es potente y fácil de usar, sin embargo en comparación al método anterior, no tenemos tanto control del formato ya que solo podemos insertar directamente la variable.

La sintaxis de un f-string es muy simple, basta con añadir una `f` antes de la comilla inicial del string y escribir el nombre de la variable a insertar rodeada por `{}` y a la hora de imprimir, Python cambiará este por el valor almacenado en su interior:

```
print(f'Hola {nombre}, ¡encantado de conocerte!')
```

```
[ ]: nombre = input('digita tu nombre: ')
     print(f'Hola {nombre}, ¡encantado de conocerte!')
```

```
digita tu nombre:  Paco
```

```
Hola Paco, ¡encantado de conocerte!
```

3.2 Características

Ahora que vimos con qué tipo de datos podemos trabajar, veamos unas de las características de Python que lo diferencian de otros lenguajes.

3.2.1 Tipado dinámico

Un lenguaje de programación tiene un sistema de tipos dinámico cuando el tipo de dato de una variable puede cambiar en tiempo de ejecución. Python efectivamente es, entonces, un lenguaje de tipado dinámico, ya que una variable puede comenzar teniendo un tipo de dato y cambiar en cualquier momento a otro tipo de dato. Por ejemplo:

[ ]:
```
a = 5
print(a)
a = "Hola mundo"
print(a)
```

```
5
Hola mundo
```

Aquí la variable `a` es creada con el valor `5`, que es un número entero (int). Luego en la tercera línea se asigna el nuevo valor `"Hola mundo"`, por lo cual el tipo de dato cambia a una cadena (str).

[ ]:
```
a = 5
print(a, type(a))
a = "Hola mundo"
print(a, type(a))
```

```
5 <class 'int'>
Hola mundo <class 'str'>
```

3.3 Tipado fuerte

Un lenguaje es de tipado fuerte cuando, ante una operación entre dos tipos de datos incompatibles, arroja un error en lugar de convertir implícitamente alguno de los dos tipos. Python es un lenguaje de tipado fuerte. Por ejemplo:

[ ]:
```
a = 5
b = "7"
print(a + b)  # ¡Error!
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-36-dda4bfe9ffd2> in <module>
      1 a = 5
      2 b = "7"
----> 3 print(a + b)  # ¡Error!

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

Aquí la tercera línea arroja un error, porque un entero (`a`) no puede sumarse a una cadena (`b`). Python podría convertir automáticamente la variable `b` a un entero o `a` a una cadena para que la operación tenga éxito; pero no lo hace, porque el sistema de tipos es fuerte. Para realizar esta operación, hay que hacer alguna conversión explícita:

[ ]:
```
a = 5
b = "7"
print(a + int(b))
```

```
12
```

[ ]:
```
a = 5
b = "7"
print(str(a) + b)
```

```
57
```

3.3.1 Multiplataforma

El intérprete de Python está disponible en multitud de plataformas (UNIX, Windows, Mac OS, etc.) por lo que si no utilizamos librerías específicas de cada plataforma nuestro programa podrá correr en todos estos sistemas sin grandes cambios.

En la proxima lección estudiaremos qué tipos de operaciones podemos realizar con los datos que soporta Python.