1. Leguanjes fuertemente tipados

En los lenguajes de programación donde es obligatorio declarar los tipos de datos de sus variables y, además, no pueden ser modificados, se los denomina **Fuertemente Tipados**. Es decir, que una vez que se declara la variable de un cierto tipo (`int`, `float`, `string`, etc.) no puede cambiarse para almacenar otro tipo de dato. En otros lenguajes de programación, las variables tiene *tipado dinámico*, les permite contener, por ejemplo, un número y luego un texto:

```
1 # Declaración e inicialización de variables en python:
2 a, b  = 5, "Programación"
3 # Se muestra el valor de a y b en pantalla:
4 print(f"{a} {b}")
5 a = "Taller de"
6 print(f"{a} {b}")
```

Como vemos, en lenguajes más flexibles se pueden declarar dos variables e inicializarlas con valores de diferentes tipos de datos, como se observa en la *línea 2* (entero y *string*). Además la variable `'a'` que inicialmente tiene almacenado el entero 5, en la *línea 5* cambia este valor numérico por un *string*, sin dar ningún tipo de error.

Este pequeño programa de `Python` produce la siguiente salida por la consola:

```
5 Programación
Taller de Programación
```

El hecho de que un lenguaje permita este tipo de asignaciones puede llevar a errores de programación en *tiempo de ejecución*, es decir, si el lenguaje no logra convertir la variable de un tipo a otro, esto producirá que el programa termine abruptamente o se "cuelgue". En cambio los lenguajes de *tipado estático* son más seguros en este sentido.

2. Reinterpretación de datos (*Typecasting*)

Como se indicó en encuentros anteriores, se puede almacenar en una variable del tipo entero un valor originalmente decimal (`float`), lo cual causa una perdida de información (el truncamiento de la parte decimal):

```
x = int(1)   # x <= 1
y = int(4.5) # y <= 4
z = int("3") # z <= 3
```

En este sentido la conversión es explícita. Pero ésta podría ser implícita, como se observa a continuación:

```
a = 3     # a-> int (implícito)
b = 5.6   # b-> float (implícito)
c = a + b # int + float -> float
```

Cuando las operaciones mezclan tipos de datos, en este caso un entero y un decimal, lo que ocurre es que el entero es "promovido" a `float` y el resultado se guarda en este tipo de dato. En caso de tener dudas sobre el tipo de dato que está almacenado en una variable se puede utilizar el comando `type()`:

```
1 a = 7   # a -> int (implícito)
2 print(type(a))
3
4 b = 3.0 # b -> float (implícito)
5 print(type(b))
6
7 print(type(a+b)) # resultad del tipo...
```

Salida

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

Es importante entender que los *strings* no tienen *casteo automático* o implícito. Eso quiere decir que si queremos hacer una operación no valida obtendremos un error:

```
1 a = 7       # a -> int
2 b = "3.0"   # b -> str
3 print(type(a+b))
```

Salida

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

3. Codificación de los datos

3.1. Declaración de Constantes

3.1.1. Constantes en binario, hexadecimal y octal

Para facilitar al programador escribir valores en otras bases el lenguaje admite que escribamos constantes en binario, hexadecimal y octal que poseen un relación de fácil conversión entre sí[1].

Por ejemplo, si se desea escribir valores representados en binario, debemos anteponer los caracteres `0b` antes de escribir la constante:

[1] Esta temática se aborda en otra materia (EFSI I).

```
a = 0b1000000000000000 # constante binaria
print(f"El binario 1000000000000000 es: {a}")
```

Esto producirá la siguiente salida en la consola:

```
El binario 1000000000000000 es: 32768
```

El mismo ejemplo podría escribirse de manera más sintética si en vez de expresar el número en binario lo expresáramos en hexadecimal anteponiendo `0x` de la siguiente manera:

```
a = 0x8000 # constante hexadecimal
print(f"El hexa 8000 es: {a}")
```

De forma menos frecuente, se podría expresar en octal anteponiendo simplemente un `0o` a la constante:

```
a = 0o100000 # constante octal
print(f"El octal 100000 es: {a}")
```

3.1.2. Caracteres

Los *strings* en `Python` están codificados en Unicode, usualmente `UTF-8`, que coincide con el código ASCII en sus primeros 127 caracteres. Existen algunos caracteres de control (no imprimibles) que son de utilidad para formatear la salida por la pantalla, aquí se enumeran algunos de ellos:

- `\n`: **Nueva Linea**. Este carácter posiciona al cursor en la linea inferior inmediata, produciendo que los próximos caracteres se escriban debajo.
- `\r`: **Retorno de carro**. Este carácter vuelve el cursor al principio de la linea sobre-escribiendo cualquiera de los caracteres escritos anteriormente en esas posiciones.
- `\t`: **Tabulador**. Este carácter deja un espacio horizontal definido por el sistema operativo. Se utiliza para encolumnar valores.
- `\b`: ***Backspace***. Este carácter produce que el cursor vuelva a la posición anterior inmediata. De esta forma se puede sobre-escribir el carácter anterior.

Nótese que estos caracteres se escriben anteponiendo el carácter \, conocido como *carácter de escape*. Éste indica que lo siguiente que se escribirá es uno de esto caracteres especiales mencionados o un carácter imprimible que no podría hacerse de otra manera a causa de la sintaxis del lenguaje, como por ejemplo las comillas dobles (") dentro de un *string* que haya empezado con las mismas.

```
print("Esto es \"encomillado doble\".")
print('Esto es \'encomillado simple\'.')
```

En el fragmento de código anterior produce la salida:

```
Esto es "encomillado doble".
Esto es ‘encomillado simple’.
```

Analice el siguiente ejemplo de uso de algunos caracteres de control y no imprimibles:

Programa 1: Uso de caracteres de control

```
1 print("Los números del 0 al 10:")
2 for i in range(0,11,1):
3   # Muestra valor de i separado por coma y una tabulación
4   print(f"{i}",end=',\t')
5
6 # reemplazamos el último ",\t" por "."
7 print("\b\b.")
```

Note que **print()** siempre agrega automáticamente un **‘\n’** al final del texto. Para evitar este comportamiento y reemplazarlo por otra cosa podemos agregar un *argumento* separando con una coma el texto y agregando **end=‘’** para indicar cómo queremos que termine la cadena.

El ejemplo anterior produce la siguiente salida:

```
0,  1,  2,  3,  4,  5,  6,  7,  8,  9,  10.
```

Si luego de salir del bucle no se volviera el cursor 2 lugares para atrás y se imprimiera el **‘.’** la salida hubiera quedado:

```
0,  1,  2,  3,  4,  5,  6,  7,  8,  9,  10,
```

Por último podemos aclarar que los caracteres pueden escribirse también en hexadecimal utilizando el carácter de escape seguido de una x (**\xNN**) u octal utilizando el carácter de escape y el número correspondiente (**\NNN**) recordando que escribiremos el valor numérico y este luego será interpretado por el sistema operativo dependiendo de la codificación correspondiente:

```
una_L = '\x4C'
una_o = '\157'
print(f"{una_L}{una_o}") # Se imprime "Lo"
```

Como ya se ha mencionado, a diferencia de otros lenguajes, en **Python** sólo existen los *strings*, no hay un tipo de dato **Character**. Sin embargo, se puede hacer una manipulación que permite *castear* un número y convertirlo a su equivalente en carácter:

```
c = 64
print(chr(c)) # Se muestra ‘@’ equivalente a 64
```

También es posible castear los números a un **str** en un formato particular con los *comandos* **bin()**, **hex()** y **oct()**:

```
1 c = 64
2 print(f"binario: {bin(c)}")
3 print(f"octal:   {oct(c)}")
4 print(f"hexa:    {hex(c)}")
```

Salida

```
binario: 0b1000000
octal:   0o100
hexa:    0x40
```

3.1.3. Strings

Por último se abarcarán los *strings* constantes y utilizar la codificación estándar de sistemas Unicode con UTF-8[2]. Por ejemplo, se pueden escribir cadenas de caracteres de la siguiente manera:

```
cadena1 = "Esto " "es un " "texto."
```

Lo cual equivale a:

```
cadena1 = "Esto es un texto."
```

Uno de los principales motivos para escribir *strings* de esta forma es la clarificación de programas que poseen grandes textos o párrafos de varios renglones, ya que no estamos obligados a que se encuentren en una misma linea.

```
print(
  "\n*** MENÚ PRINCIPAL ***\n\n"
  "\t1. Opción A\n"
  "\t2. Opción B\n"
  "\t3. Opción C\n"
  "\t4. Opción D\n"
  "\nIngrese una opción:", end=' ')
```

La única sentencia que hay en este fragmento de código produce la siguiente salida por la pantalla:

```
*** MENÚ PRINCIPAL ***

    1. Opción A
    2. Opción B
    3. Opción C
    4. Opción D

Ingrese una opción: _
```

Esto ayuda al los programadores a tener una mejor idea de cómo se ve el *string* en la pantalla, ya que sería engorroso verlo en una sola linea y es más eficiente que hacer un `print()` por cada linea.

Como ya se ha mencionado **Python** codifica por default en Unicode, que es el más habitual en Sistemas Operativo moderons que son *Tipo-UNIX*, por ejemplo, GNU/Linux y MacOS, utilizan para la codificación de los textos por *default* en UTF-8. Sabiendo esto, si se realiza un software para este tipo de sistemas, es posible imprimir cualquier carácter del código correspondiente a éstos. Puede utiliza directamente el código del carácter únicode:

[2]Esta temática se aborda en otra materia (EFSI I).

```
print("\u00A2") # Centavos U+00A2 (también funciona chr(0xA2))
print("\u20AC") # Euros    U+20AC (también funciona chr(0x20AC))
```

El fragmento de programa anterior muestra por pantalla los caracteres '¢' y '€' respectivamente.

Pero si se deseara utilizar la codificación, propiamente dicha, y teniendo en cuenta que en UTF-8 los códigos son variables, podremos hacer lo siguiente:

```
cents = b"\xC2\xA2"          # 'bytes' (anteponiendo b al str)
print(cents.decode('utf-8')) # imprime el caracter centavos.
```

Cuando escribimos una **'b'** antes del *string* significa que cada carácter que pongamos ocupará un **byte**, es decir que en realidad estamos guardando una *cadena de bytes*. Esto es útil cuando se desea manejar valores binarios o sin codificación.

3.2. Valores Constantes

Hay software que tiene o necesita valores fijos (cantidad de piezas, cuadros en un tablero, movimientos en una partida, etc.). Es deseable evitar que estos números aparezcan como literales en medio del código donde no es claro de donde vienen o cuál es su función real. A éstos se los denomina **números mágicos** (*magic numbers*) y es deseable que no estén presentes en el código.

```
if cant_clientes < 500: # magic number...¿qué significa?
    # ...
```

En Python no existen variables con valores constantes, pero por convención se interpreta que si el nombre de la variable está escrito totalmente en mayúsculas, este contiene una "constante":

```
MAX_CLIENTES = 500 # Constante por convención
# ...
if cant_clientes < MAX_CLIENTES:
  # ...
```