Introducción a *Python*

Instrucciones de uso

A continuación, se presenta la sintaxis básica del lenguaje de programación Python junto con ejemplos interactivos.

Variables y tipos de variables

Podemos entender una variable como un contenedor en el que guardamos datos para poder manejarlos más adelante. En Python, las variables no tienen un tipo fijo, es decir, no es necesario especificar si la variable será numérica, un carácter, una cadena de caracteres o una lista, por ejemplo. Además, las variables pueden ser declaradas e inicializadas en cualquier momento, a diferencia de otros lenguajes de programación.

Para declarar una variable, utilizamos la expresión `nombre_de_variable = valor`. Se recomienda revisar el documento PEP-8 para seguir buenas prácticas en la nomenclatura de variables. Sin embargo, en términos generales, se recomienda evitar el uso de mayúsculas al inicio, separar las palabras con el carácter `_` y no utilizar acentos ni caracteres especiales como el símbolo del `€` o la `ñ`, por ejemplo.

Veamos algunos ejemplos de declaración de variables y cómo usarlas:

```
# Declaramos una variable llamada 'variable_numerica' que contiene el
valor entero 12.
variable_numerica = 12

# Declaramos una variable llamada 'monstruo' que contiene el valor
'Godzilla'.
monstruo = 'Godzilla'

# Declaramos una variable llamada 'planetas' que es una lista de
cadenas de caracteres.
planetas = ['Mercurio', 'Venus', 'Tierra', 'Marte']

la_mi_edad = 25
la_mi_edad_en_5 = la_mi_edad + 5
# 'Imprimimos' el valor calculado que será, efectivamente, 30
print(la_mi_edad_en_5)
```

```
30
```

Los tipos de datos nativos que una variable en Python puede contener son: números enteros (`int`), números decimales (`float`), números complejos (`complex`), cadenas de caracteres (`string`), listas (`list`), tuplas (`tuple`) y diccionarios (`dict`). Veamos cada uno de estos tipos:

```
# Un número entero
int_var = 1
another_int_var = -5
# Podemos sumarlos, restarlos, multiplicarlos o dividirlos.
print(int_var + another_int_var)
print(int_var - another_int_var)
print(int_var * another_int_var)
print(int_var / another_int_var)

# También podemos realizar la división entera.
# Dado que solo trabajamos con números enteros, no habrá parte
decimal.
print(int_var // another_int_var)
```

```
-4
6
-5
-0.2
-1
```

El comportamiento del operador `/` es una de las diferencias entre Python 2 y Python 3. Mientras que en Python 3 el operador `/` realiza la división real entre dos números enteros (fíjate que `1 / -5` da como resultado `0.2`), en Python 2 realizaba la división entera (por lo que el resultado de ejecutar `1 / -5` en Python 2 sería `-1`). En Python 3, para realizar la división entera, se usa el operador `//`.

```
# Un número decimal o 'float'
float_var = 2.5
another_float_var = .7
# Convertimos un número entero en uno decimal utilizando la función
'float()'
encore_float = float(7)
# Podemos hacer lo mismo en sentido contrario con la función 'int()'
new_int = int(encore_float)

# Podemos realizar las mismas operaciones que con los números enteros,
pero en este caso la división será
# decimal si alguno de los números es decimal.
print(another_float_var + float_var)
print(another_float_var - float_var)
print(another_float_var * float_var)
print(another_float_var / float_var)
print(another_float_var // float_var)  # La división entera con
números flotantes puede no ser intuitiva.
```

```
3.2
-1.8
1.75
```

```
0.27999999999999997
0.0

# Un número complejo
complex_var = 2 + 3j
# Podemos acceder a la parte imaginaria o a la parte real:
print(complex_var.imag)
print(complex_var.real)

3.0
2.0

# Cadena de caracteres
my_string = 'Hola, Bio! ñç'
print(my_string)

Hola, Bio! ñç
```

Observa que podemos incluir caracteres Unicode (como `ñ` o `ç`) en las cadenas. Esto también es una novedad de Python 3, ya que las variables de tipo `str` ahora utilizan codificación UTF-8.

```
# Podemos concatenar dos cadenas utilizando el operador '+'.
same_string = 'Hola, ' + 'Bio' + '!' + ' ñc'
print(same_string)

# En Python también podemos utilizar comodines (wildcards) como en la
función 'sprintf' de C. Por ejemplo:
name = "Guido"
num_emails = 5
print("Hola, %s! Tienes %d nuevos correos" % (name, num_emails))

Hola, Bio! ñc
Hola, Guido! Tienes 5 nuevos correos
```

En el ejemplo anterior, hemos sustituido en el *string* la cadena `%s` por el contenido de la variable `name`, que es un *string*, y `%d` por `num_emails`, que es un número entero. También podríamos utilizar `%f` para números decimales (podríamos indicar la precisión, por ejemplo, con `%5.3f`, donde el número tendría un tamaño total de cinco dígitos y tres serían para la parte decimal). Hay muchas otras posibilidades, pero debemos tener en cuenta el tipo de variable que queremos sustituir. Por ejemplo, si utilizamos `%d` y el contenido es un *string*, Python generará un mensaje de error. Para evitar esta situación, se recomienda usar la función `str()` para convertir el valor a *string*.

También podemos mostrar el contenido de las variables sin especificar su tipo utilizando `format`:

```
print("Hola, {}! Tienes {} nuevos correos".format(name, num_emails))

Hola, Guido! Tienes 5 nuevos correos
```

Ahora presentaremos otros tipos de datos nativos más complejos: listas, tuplas y diccionarios:

```
# Definimos una lista con los nombres de los planetas (como
_strings_).
planets = ['Mercurio', 'Venus', 'Tierra', 'Marte',
           'Júpiter', 'Saturno', 'Urano', 'Neptuno']
# También puede contener números.
prime_numbers = [2, 3, 5, 7]

# Una lista vacía
empty_list = []

# O una mezcla de cualquier tipo:
sandbox = ['3', 'una cadena', ['una lista dentro de otra lista',
'segundo elemento'], 7.5]
print(sandbox)
```

```
['3', 'una cadena', ['una lista dentro de otra lista', 'segundo
elemento'], 7.5]
```

```
# Podemos añadir elementos a una lista.
planets.append('Plutón')
print(planets)
```

```
['Mercurio', 'Venus', 'Tierra', 'Marte', 'Júpiter', 'Saturno',
'Urano', 'Neptuno', 'Pluto', 'Plutón']
```

```
# O podemos eliminar elementos.
planets.remove('Plutón')
print(planets)
```

```
['Mercurio', 'Venus', 'Tierra', 'Marte', 'Júpiter', 'Saturno',
'Urano', 'Neptuno', 'Pluto']
```

```
# Podemos eliminar cualquier elemento de la lista.
planets.remove('Venus')
print(planets)
```

```
['Mercurio', 'Tierra', 'Marte', 'Júpiter', 'Saturno', 'Urano',
'Neptuno', 'Pluto']
```

```
# Siempre que añadamos un elemento, será al final de la lista. Una
lista está ordenada.
planets.append('Venus')
print(planets)
```

```
['Mercurio', 'Tierra', 'Marte', 'Júpiter', 'Saturno', 'Urano',
'Neptuno', 'Pluto', 'Venus']
```

```
# Si queremos ordenarla alfabéticamente, podemos utilizar la función
'sorted()'
print(sorted(planets))
```

```
['Júpiter', 'Marte', 'Mercurio', 'Neptuno', 'Pluto', 'Saturno',
'Tierra', 'Urano', 'Venus']

# Podemos concatenar dos listas:
monsters = ['Godzilla', 'King Kong']
more_monsters = ['Cthulhu']
print(monsters + more_monsters)

['Godzilla', 'King Kong', 'Cthulhu']

# Podemos concatenar una lista con otra y guardar el resultado en la
misma lista:
monsters.extend(more_monsters)
print(monsters)

['Godzilla', 'King Kong', 'Cthulhu']

# Podemos acceder a un elemento específico de la lista:
print(monsters[0])
# El primer elemento de una lista es el 0, por lo tanto, el segundo
será el 1:
print(monsters[1])
# Podemos acceder al último elemento usando índices negativos:
print(monsters[-1])
# El penúltimo elemento:
print(monsters[-2])

Godzilla
King Kong
Cthulhu
King Kong

# También podemos obtener partes de una lista utilizando la técnica de
'slicing'.
planets = ['Mercurio', 'Venus', 'Tierra', 'Marte',
           'Júpiter', 'Saturno', 'Urano', 'Neptuno']
# Por ejemplo, los dos primeros elementos:
print(planets[:2])

['Mercurio', 'Venus']

# O los elementos entre las posiciones 2 y 4
print(planets[2:5])

['Tierra', 'Marte', 'Júpiter']
```

Observa el último ejemplo: en la posición 2 encontramos el tercer elemento de la lista (`'Tierra'`), ya que las listas comienzan a indexarse en 0. Además, el último índice indicado (la posición `5`) no se incluye en el resultado.

```
# O los elementos desde el segundo hasta el penúltimo:
print(planets[1:-1])
```

```
['Venus', 'Tierra', 'Marte', 'Júpiter', 'Saturno', 'Urano']
```

La tècnica de ***slicing*** és molt important i ens permet gestionar llistes d'una manera molt senzilla i potent. Serà imprescindible dominar-la per afrontar molts dels problemes que haurem de resoldre en el camp de la ciència de dades.

```
# Podemos modificar un elemento específico de una lista:
monsters = ['Godzilla', 'King Kong', 'Cthulhu']
monsters[-1] = 'Kraken'
print(monsters)
```

```
['Godzilla', 'King Kong', 'Kraken']
```

```
# Una tupla és un tipus molt semblant a una llista, però és immutable,
és a dir, un cop declarada
# no podem afegir-hi elements ni eliminar-ne:
birth_year = ('Stephen Hawking', 1942)
# Si executem la línia següent, obtindrem un error de tipus
'TypeError'
birth_year[1] = 1984
```

Los errores en Python suelen ser muy informativos. Una búsqueda en internet nos ayudará en la gran mayoría de los problemas que podamos encontrar.

```
# Una cadena de caracteres también se considera una lista de
caracteres.
# Por lo tanto, podemos acceder a una posición determinada (aunque no
modificarla):
name = 'Albert Einstein'
print(name[5])

# También podemos usar slicing con las cadenas de caracteres
print(name[7:15])

# Podemos separar la cadena en base al carácter que elijamos (en este
caso, el espacio en blanco) utilizando
# la función 'split()'.
n, surname = name.split()
print(surname)

# Y podemos convertir una cadena en una lista de caracteres
fácilmente:
chars = list(surname)
print(chars)

# Para unir los diferentes elementos de una lista mediante un
carácter, podemos utilizar la función
```

```
# 'join()':
print(''.join(chars))
print('.'.join(chars))
```

```
t
Einstein
Einstein
['E', 'i', 'n', 's', 't', 'e', 'i', 'n']
Einstein
E.i.n.s.t.e.i.n
```

```
# El operador ',' crea tuplas. Por ejemplo, el típico problema de asignar el valor de una
# variable a otra en Python se puede resolver en una línea de manera muy elegante utilizando
# tuplas (es un idiomático):
a = 5
b = -5
a, b = b, a
print(a)
print(b)
```

```
-5
5
```

El ejemplo anterior es un *idioma* típico de Python. En la tercera línea, creamos una tupla `(a, b)` a la que asignamos los valores uno por uno de la tupla `(b, a)`. Los paréntesis no son necesarios, y por eso la notación queda tan reducida.

Finalmente, presentaremos los diccionarios, una estructura de datos muy útil en la que asignamos un valor a una clave en el diccionario:

```
# Códigos internacionales de algunos países. La clave o 'key' es el código del país, y el valor, su nombre:
country_codes = {34: 'España', 376: 'Andorra', 41: 'Suiza', 424: None}

# Podemos buscar
my_code = 34
country = country_codes[my_code]
print(country)
```

```
España
```

```
# Podemos obtener todas las claves:
print(country_codes.keys())
```

```
dict_keys([34, 376, 41, 424])
```

```
# O los valores:
print(country_codes.values())
```

```
dict_values(['España', 'Andorra', 'Suiza', None])
```

Es muy importante notar que los valores que obtenemos de las claves o al imprimir un diccionario no están ordenados. Es un error muy común suponer que el diccionario se guarda internamente en el mismo orden en que fue definido, y es una fuente habitual de errores no tenerlo en cuenta.

```
# Podemos modificar valores en el diccionario o agregar nuevas claves.

# Definimos un diccionario vacío. 'country_codes = dict()' es una
notación equivalente:
country_codes = {}

# Añadimos un elemento:
country_codes[34] = 'España'

# Añadimos otro:
country_codes[81] = 'Japón'

print(country_codes)
```

```
{34: 'España', 81: 'Japón'}
```

```
# Modificamos el diccionario:
country_codes[81] = 'Andorra'

print(country_codes)
```

```
{34: 'España', 81: 'Andorra'}
```

```
# Podemos asignar un valor vacío a un elemento:
country_codes[81] = None

print(country_codes)
```

```
{34: 'España', 81: None}
```

Els valors buits ens seran útils per declarar una variable que no sapiguem quin valor o quin tipus de valor contindrà i per fer comparacions entre variables. Habitualment, els valors buits són *None* o '', en el cas de les cadenes de caràcters.

```
# Podem assignar el valor d'una variable a una altra. És important que
s'entenguin les
# línies següents:
a = 5
b = 1
print(a, b)
# b conté la 'direcció' del contenidor al qual apunta 'a'.
b = a
print(a, b)
```

```
# Vegem ara què passa si modifiquem el valor d'a o b:
a = 6
print(a, b)
b = 7
print(a, b)
```

Fins aquí hem presentat com declarar i utilitzar variables. Recomanem la lectura de la documentació oficial en línia per a fixar els coneixements explicats.