Introducción a Python

Contenido

¿Qué es Python?

Python es un lenguaje de programación **interpretado multiparadigma** con un **tipado dinámico**.

Suena a chino, ¿cierto? No te preocupes que todo esto se va a explicar de la forma más simple posible.

Es interpretado

En la programación tenemos dos tipos de lenguajes, principalmente, los **interpretado** y los **compilados**. Esto no es más que agruparlos según su forma de ser traducidos de "lenguaje humano" a lenguaje máquina.

Los lenguajes más clásicos, es decir, los compilados, necesitan ser compilados o "**traducidos**" antes de poder ser ejecutados. Tu escribes tu código y para probar su funcionamiento debes compilar y después ejecutar el resultado. Esto obviamente es más lento, pero tiene el beneficio de que el ejecutable final es más eficiente. Lenguajes clásicos compilados serían, por ejemplo, C, C++, Java….

En cambio, los lenguajes interpretados, son "ejecutados sobre la marcha". Por decirlo de una manera simple, son traducidos según se va leyendo. Esto aporta una ventaja clave sobre los compilados, multiplataforma. Mientras una máquina tenga el intérprete, puede ejecutar el código, a diferencia del otro caso que debe ser compilado en cada máquina primero.

Por ponerlo simple: Tu tienes un programa escrito en C y quieres que tu amigo, Larry, lo pruebe en su ordenador. Larry tiene que coger tu código, escribir el comando de compilación y después ejecutar lo que ha generado. Si lo hacemos en el otro caso, Larry puede coger tu código y ejecutarlo directamente. Parece que te ahorras solo un paso, pero ese paso puede ser más de 40 minutos mirando una pantalla, lo digo por experiencia. Compila un código con más de 3GB de código fuente y verás lo que es tardar.

Otra ventaja que ofrece un código interpretado es que puedes "escribir sobre la marcha". Esto es lo primero que haremos una vez entremos en materia. ¿Esto que significa? Puedes abrir el interprete e ir escribiendo el código a la vez que se ejecuta, por lo que vas viendo "en vivo" lo que está pasando.

Dentro de esta categoría tenemos algunos como Python, JavaScript/TypeScript, Ruby…

Multiparadigma

Un **paradigma** no es más que un modelo de desarrollo, dicho de otra forma, para lo que vale el lenguaje. Los paradigmas que soporta Python son los mismos que C++, estos son los paradigmas **Imperativo**, **Funcional y Orientado a objetos**.

Imperativo

Que sea imperativo significa, sin complicar demasiado la respuesta, que se describe paso a paso el conjunto de instrucciones que deben ejecutarse para solucionar un problema. Casi todos los que vamos a trabajar o conocéis son así, ignoremos por completo los lenguajes **declarativos.**

Funcional

Por ponerlo de forma sencilla, tienes una **función definida** que siempre va a actuar igual, sean cuales sean las condiciones, que dará una salida esperada y predecible. Por poner un ejemplo, tienes la operación matemática de la suma (+) y no importa que términos quieras sumar, sabes exactamente que va a hacer esa función. Tienes dos números de entrada y un número de salida que es el resultado de la adición o sustracción de los de entrada.

Orientada a Objetos (POO)

La programación orientada a objetos ofrece la particularidad en la forma de obtener los resultados. Los objetos manipulan las entradas para la obtención de un resultado específico.

También nos permite la agrupación en bibliotecas o librerías, así como crear las nuestras propias.

Traduciendo a lenguajes común, podemos crear clases para agrupar funciones específicas y usar clases de librerías para facilitarnos la vida.

Un **objeto** no sería más que una colección de funciones y variables específicas con una idea en mente. Por poner un ejemplo simple, tenemos el objeto coche que tiene en sus variables el color, modelo y potencia y en sus funciones arrancar, parar o acelerar. Sabes lo que puede hacer el coche y como debe interactuar con otros objetos.

Es un concepto raro de entender, pero es de lo más útil.

De tipado dinámico

Este puede ser el concepto más "complicado" de entender para los que vienen de lenguajes de **fuerte tipado**, como lo son todas las variantes de C.

Más que complicado de entender, de aplicar al ser tan alejado de la costumbre.

Definiremos las variables de Python como **cajas de tamaño variable**. Tu en una caja puedes guardar manzanas y ahora tendrías una caja de manzanas, pero sigue siendo una caja. Ahora la caja la vacías y la llenas de engranajes, no tiene nada que ver con manzanas ahora, pero la caja sigue siendo una caja. Con esto quiero decir que puedes poner cualquier tipo de objeto en una variable y justo en la línea siguiente poner algo completamente distinto que seguirá funcionando.

Ahora dicho con lenguaje más de programación, las variables no son más que contenedores dinámicos en los que su tipo puede ser asignado y reasignado continuamente a lo largo de un script.

Instalando Python

A partir de aquí vamos a ir trabajando con el código real, para ello necesitamos nuestro interprete.

Instalar Python en Windows es realmente simple, sólo tienes que ir a la web de descargas de Python y descargar la última versión.

Si durante la instalación te pregunta si debe instalar Pip, di que sí, al igual que si dice "añadir a Path".

Listo, tu ordenador ya tiene el interprete de Python y puedes ejecutar código libremente.

Si estás trabajando desde Linux, Python debe venir por defecto, pero asegúrate que tengas Python 3 instalado, que lo normal es que sí. Lo que deberás instalar por tu cuenta será Pip, hazlo según tu sistema operativo.

En sistemas basados en Debian (como Ubuntu, Raspbian o Linux Mint): *sudo apt install pip*

Por defecto te cogerá la versión de Pip de Python 3.

Sintaxis de Python

La **sintaxis** de un lenguaje no es más que el orden y uso de palabras para dar una instrucción correcta. Por poner un ejemplo en nuestro idioma:

- Charlie hola soy. Aunque las palabras son correctas, el orden de ellas está mal, sintaxis incorrecta.
- Hola são =2 Charlie. El orden podríamos decir que es correcto, pero ahí hay una palabra en otro idioma, por lo que también es una sintaxis incorrecta.
- Hola soy Charlie. Ahora sí, tanto el orden como las palabras son correctas, sintaxis correcta.

Es un ejemplo tonto, pero es una manera simple de explicar la importancia a la hora de escribir, en especial las condiciones.

Si ya sabes programar, esto es fácil, la sintaxis de Python es, prácticamente, la misma que se usa en C++, ya que Python en realidad está hecho en ese lenguaje y basado directamente en él. Su interprete está hecho íntegramente en C++.

Pero, entonces, ¿Para qué una sección sobre la sintaxis? Fácil, es igual pero no, **la indentación es la cuestión**. En C++ tú puedes escribir si quieres todo el código en una sola línea simplemente separando las instrucciones con punto y coma y enmarcando funciones con las llaves. En Python esto no es posible, para empezar porque no hay ni llaves ni punto y coma.

Todo debe estar correctamente indentado, esto no es más que aplicar el correcto sangrado a cada línea cuando corresponde. Pongamos otro ejemplo en un **protolenguaje de programación**.

```
LAURA = CHICA
COMIDA = VERDURA
IF COMIDA IS NOT VERDURA:
LAURA NO SE QUEJA
IF COMIDA IS PIZZA:
LAURA FELIZ
ELSE:
LAURA SE QUEJA
```

Como puedes ver en este pequeño segmento, cada línea dentro de la condición está a un espacio a la derecha de esta. Esto significa que, si se cumple la condición, se hará todo lo que está por debajo y a la derecha de esta.

Esto obliga al programador a hacer si o si códigos más legibles y bien formateados, a diferencia de algunas cosas que se pueden llegar a ver en otros lenguajes.

No te preocupes si no lo entiendes ahora, todo será más fácil de ver con ejemplos.

Gracias a este ejemplo podemos explicar lo siguiente, los tipos de instrucción. Es muy simple, solo tenemos **instrucciones simples e instrucciones complejas**. Las simples es que solo realizan una acción y las otras "abren" un conjunto de instrucciones. Poniendolo claro, si una instrucción acaba la línea con un ':' es una instrucción compleja, si no, es una simple.

"Laura = chica" sería una instrucción simple, mientras que "if comida is not verdura" una compleja, si la condición se cumple da pie a que se ejecuten dos más, al menos, dentro de su bloque.

Pasemos ya a la chicha del tema.

Variables y tipos

Como hemos dicho antes, las variables son contenedores para almacenar información, pero ahora necesitamos saber qué clase de información podemos almacenar.

Distinguimos dos tipos principales de datos:

- Datos **simples**:
  - **Numéricos**
  - **Booleanos**
- Datos **compuestos o estructurados**:
  - **Listas:**
    - **Array**
    - **Tuple (tuplas)**
    - **String**
    - **Unicode**
    - **Xrange**
    - **Range**
  - **Mapeos: Diccionarios** (igual que el Json en otros lenguajes)
  - **Conjuntos: Set, frozenset** (realmente muy similares al diccionario)
  - **Otros:**
    - **File**
    - **None**
    - **Date**
    - **Datetime**
    - **…**

Datos simples

Por decirlo de una manera, valga la redundancia, simple, solo almacenan una cosa.

Numéricos

No creo que haga falta explicar esto, un número. Puede ser de tipo entero (**int**), sin parte decimal, o con coma flotante (**float**), con decimales.

Booleanos

El si o no, verdadero o falso, literalmente en código **True** o **False.**

Esto viene bien para las condiciones y muchas veces se va a usar de forma indirecta, dentro de, por ejemplo, comprobaciones de igualdades.

Datos estructurados

Obviamente hay muchos más de los que aquí os pongo y ni siquiera me merece la pena explicar algunos de los que has visto antes porque, básicamente, no los vamos a usar o rara vez lo usarás.

String

Este es un poco confuso ponerlo aquí, porque realmente solo almacena "una cosa", una cadena de caracteres, pero claro, ya son varios caracteres.

Por ponerlo fácil, sería un tipo de variable que almacena texto.

Este tipo, además, viene con funciones propias, es un **objeto** (el primero que vemos). ¿Qué significa? Que tiene funciones definidas para facilitar las cosas. Esas funciones las podrías crear tu a mano, pero ya existen y las coges directamente. Una función por ejemplo sería para cortar por un punto el texto, encontrar una palabra, quitar todos los caracteres iguales, pasarlo entero a mayúsculas o minúsculas… Realmente muchas funciones útiles.

Array y tupla

Lo mismo, pero no. Ambos son listas de objetos o variables, que pueden ser tanto simples como estructuras, con la principal diferencia en que los arráis son dinámicos y las tuplas estáticas, uno se puede modificar cuando quieras y el otro tal cual se crea se queda.

Un array se declara de esta forma:

```
Cosas = ['llaves', 'piedras', 'pulsera', 'lampara']
```

Mientras que el tuple

```
Animales = ('perro', 'gato', 'mono', 'loro')
```

Se ven similares, porque casi que lo son, pero con el array podrías hacer esto:

```
Cosas.append('albóndiga')
```

Y la lista quedaría

```
['llaves', 'piedras', 'pulsera', 'lampara',' albóndiga']
```

Mientras que con la otra clase no podrías.

Al igual que con el string, se tratan de **objetos** y tienen funciones asociadas (como append que acabas de ver).

Diccionarios

Al igual que en la mayoría de los lenguajes, tenemos algo similar a las estructuras. Python no es distinto, pero en sí sus "estructuras" sí lo son.

Cuando hablamos de un diccionario real, tenemos una palabra y su definición, llevando a Python, tendríamos pares de **clave y valor**.

Por ponerlo simple, piensa en un libro con una colección de **keys,** que serían las entradas de un diccionario, y su **value** que sería la definición. Y, al igual que con nuestra lengua, el diccionario puede ser modificado. Añadir entradas, modificar su significado o, incluso, borrarlas.

Al lio, ¿Cómo se usan? Se declararían de la siguiente manera:

```
PERSONA1 = {
    'NOMBRE': 'JOSÉ LUÍS',
    'EDAD': 68,
    'HIJOS': 4,
    'NOMBEHIJOS': ['PACO', 'JORGE', 'BARBARA', 'ALEJANDRA']
}
```

Pero ¿qué vemos aquí? ¿Varios tipos distintos? ¿Asignar con ':' en lugar de usar el '='?, así es, puedes usar todos los tipos que te apetezca, incluso diccionarios dentro de diccionarios.

También se trata de un objeto, a si que tiene sus funciones.

¿Cómo modificamos un valor? Tal cual así:

```
Persona1['edad'] = 69
```

De esta forma también podemos sacar su valor (más adelante veremos como hacerlo de forma segura).

También se puede añadir nuevas entradas de la misma forma, no hace falta declararlas previamente.

Todas las funciones en detalle las veremos más adelante, no solo de los diccionarios, también de las demás **clases**.

Condiciones

Las condiciones son lo principal con lo que se suele trabajar. Por ponerlo de una manera simple, son preguntas de **si** y **no**. Te pongo un ejemplo en nuestro lenguaje:

```
¿Sí tengo 10 manzanas? -> Me como una.
Pero si no y ¿tengo más de 5? -> Compro hasta tener 10.
Pero si no -> Compro directamente 10.
```

Fácil de entender, ¿no?

Pero debemos ahora pasarlo a código, aunque antes tenemos que ver como comprobar nuestras condiciones, para eso usamos los **operadores**.

Operadores

Primero de nada, los operadores simples o comparadores.

Operador	Función	Ejemplo	Resultado
==	Igual que	10 == 10 10 == 11 'Queso' == 'Pizza'	**True** **False** **False**
!=	Distinto de	10 != 10 10 != 11 'Queso' != 'Pizza'	**False** **True** **True**
<	Menor que	10 < 11 10 < 8 'Queso' < 'Pizza'	**True** **False** **False**
>	Mayor que	10 > 11 10 > 8 'Queso' > 'Pizza'	**False** **True** **True**
<=	Menor o igual	10 <= 11 10 <= 10 'Queso' <= 'Pizza'	**True** **True** **False**
>=	Mayor o igual	10 >= 11 10 >= 10 'Queso' >= 'Pizza'	**False** **True** **True**

¿Comparando strings? ¿Qué es esto? Se puede sin problema. En el caso de los comparadores mayor o menor que, lo que te compara sería por orden alfabético. En los ejemplos vemos una palabra que empieza por 'Q' y otra por 'P', la 'Q' es mayor que la 'P' en el alfabeto.

Pues pasarlo a código es aún más simple, tan solo es casi como traducir al inglés. Vamos a ver nuestro primer código completo:

```
manzanas = 0

if manzanas == 10:
    manzanas = manzanas - 1
elif manzanas >= 5:
    while manzanas < 10:
        manzanas = manzanas + 1
else:
    manzanas = 10

print(manzanas)
```

Estas viendo un bucle **while** ahí, pero de momento lo ignoramos hasta el siguiente tema.

¿Cómo crees tu que funcionaría este código? ¿Cuál será el resultado que sacará ese **print**?

Obviamente, al ser una sola iteración y viendo como manzanas empieza valiendo 0, entrará directamente en la última opción, la “por defecto”.

Importante en Python no existen los **switch** como en la mayoría de lenguajes.

Pero ¿Qué pasa si quiero poner más de una condición por cada **if**? Para esto tenemos los operadores.

Operador	Función	Ejemplo	Resultado
and	Si y solo si todos los elementos son **True** dará como resultado **True**. Si no **False**	**True** and **True** **False** and **False** **True** and **False**	**True** **False** **False**
or	Si algún elemento es **True** el resultado será **True**, si no **False**	**True** and **True** **False** and **False** **True** and **False**	**True** **False** **True**
not	Este operador solo afecta a un elemento y dará el valor contrario	not **True** not **False**	**False** **True**

Veamos un ejemplo:

```
manzanas = 0

if manzanas == 10 or manzanas != 20:
    manzanas = manzanas - 1
elif manzanas >= 5 and manzanas < 9:
    while manzanas < 10:
        manzanas = manzanas + 1
else:
    manzanas = 10

print(manzanas)
```

¿Ahora que se mostrará por pantalla?

Bucles

¿Qué pasa si queremos hacer una acción múltiples veces? Para eso tenemos los bucles.

En Python solo existen 2 tipos de bucle, aunque realmente para que queremos más.

For

El bucle para **iterar**. ¿Qué significa esto? Que se usa para recorrer algo especifico, podríamos traducirlo como “haz esto durante…” o “recorre estos objetos”.

Se usan y declaran de forma diferente a lo que son en los lenguajes comunes, lo más parecido al clásico “for(int i = 0; i < 10; i++)” de C deberíamos hacerlo así:

```
for i in range(10):
    print(i)
```

Con la función **range**(stop) creamos un, como bien dice, desde 0 hasta el número indicado en el parámetro **stop**, este no incluido. Si ejecutas el código, verás por pantalla "0 1 2 3 4 5 6 7 8 9", aunque en diferente línea, el **print()** siempre mete un salto de línea al final.

Pero ¿sólo nos sirve para números? Ni mucho menos. Con El for puedes recorrer listas, strings, diccionarios…

```
lista = [1,2,3,4,5]
string = 'abcdefg'
diccionario = {'porp1': 1, 'prop2': 2, 'prop3': 3}

for i in lista:
    print(i)
# Mostrara 1 2 3 4 5
for c in string:
    print(c)
# Mostrara a b c d e f g
for k in diccionario.keys():
    print(k)
# Mostrara prop1 prop2 prop3
for v in diccionario.values():
    print(v)
# Mostrara 1 2 3
for (k,v) in diccionario.items():
    print(k,v)
# Mostrara prop1 1 prop2 2 prop3 3
```

Ves que se pueden crear hasta más de una variable que iteran a la vez, como es el caso del ultimo for, tenemos la variable k y v que iteran cada uno una de las propiedades que devuelve la función **items**() de los diccionarios.

While

Podríamos traducir este bucle como "mientras que se cumpla esto, haz esto", pongamos un ejemplo simple:

```
entrada = ''

while(entrada != 'f'):
    print('Da una letra por teclado:')
    entrada = input()
```

En este caso, en cada **iteración** del bucle se espera una entrada por teclado con la función **input()** (en la terminal escribes y le das al **enter**, de esta forma podemos interactuar con nuestro código). El código te seguirá pidiendo la letra hasta que no se cumpla la condición de que la letra que hayas metido sea la 'f'.

Las condiciones que puedes poner son las mismas que las que hemos visto en el tema anterior.

Con esto podemos crear un sinfín de códigos, desde un simple menú hasta realizar cálculos super complejos.

Control de bucles

Tenemos ciertas palabras clave, que podemos decir que son como operadores, que nos dan un control extra sobre nuestros bucles.

Imagina, por ejemplo, que quieres buscar un valor específico dentro de una lista, pero claro, lo encuentras y aun te queda más de la mitad de la lista que recorrer. Lo normal sería que llegase hasta el final antes de acabar, pero qué pérdida de tiempo, ¿no?, lo suyo sería salir del bucle en ese momento y ahorrar ese tiempo de ejecución innecesario.

Otro caso, no quieres que se ejecute nada de lo que hay a continuación si se cumple cierta condición, pero no quieres salir del bucle.

Y, por último, ¿qué pasa si cuando se cumpla cierta condición quieres que el código no haga absolutamente nada?

Para eso tenemos estos operadores.

Break

Esta instrucción se utiliza para finalizar un bucle, es decir, si ejecutas el **break**, saldrás del bucle de inmediato y continuarás con el resto del código.

Continue

La instrucción continue obliga al bucle a ir a la siguiente iteración ignorando todo lo que quedase por ejecutar del mismo.

Pass

Como bien indica su nombre, no hace nada, pasa. Esta instrucción se puede escribir también como '**…**' y no tiene por qué ser usada dentro de un bucle, puede usarse en cualquier ámbito.

```
for i in range(100):
    if i > 10:
        break       # Si llega a 10, salimos del código
    if i < 5:
        print(i)
        continue    # Hace que las siguientes lineas se ignoren
    if i >= 5:
        pass        # No hace nada y continua el bucle
    print("Fuera de condiciones")
```

En el ejemplo, veremos como salen los números del 0 al 4 seguido de varios "Fuera de condiciones" y cuando el iterador llegue a 10, finalizará la ejecución.

Con esto finalizaríamos la parte básica de Python.