Iniciación a Python

Ajuntament de Barcelona

Barcelona Activa

Índice

1 QUÉ ES PYTHON Y CÓMO EMPEZAR A PROGRAMAR

Empezaremos descubriendo los rasgos más distintivos de Python y viendo sus aplicaciones más relevantes. Hay muchos lenguajes de programación, pero Python se ha convertido en uno de los más populares en todo el mundo.

Después de la introducción al lenguaje, pasaremos a instalar el intérprete, el responsable de que nuestro código sea entendido por nuestro ordenador.

Más tarde, instalaremos paso a paso el entorno de programación que nos facilitará la gestión de nuestro código.

Con el intérprete y el entorno de programación ya tendremos todas las herramientas para empezar a programar. Solo nos hará falta tener ganas de aprender y, quizás, en ciertos momentos, un poco de paciencia si las cosas no salen a la primera. ¡El mundo de Python nos espera!

1.1 QUÉ ES PYTHON Y QUÉ APLICACIONES TIENE

¡Os damos la bienvenida al mundo de Python!

¿te gusta el desarrollo web y quieres crear aplicaciones? ¿Has pensado alguna vez en diseñar un videojuego? ¿Te interesa el *big data* y la inteligencia artificial?

Sí, todo esto se puede crear con Python. Si tienes imaginación y ganas de aprender a programar con este lenguaje, un día podrás llegar a crear aquello que imagines o encontrar aquel trabajo con el que tanto habías soñado.

Así que no perdamos más tiempo y empecemos a conocer qué es realmente Python. Vamos, entonces, a destacar sus características principales que lo hacen tan popular:

- Es un lenguaje de código libre. Puedes modificar o mejorar aquello que quieras del código fuente.

- Está basado en la orientación a objetos. Tu código se escribirá en bloques reaprovechables que tendrán funciones y variables que podrás crear siguiendo tu propio criterio, siempre y cuando cumplas una serie de normas.

- Fácil de aprender. Python se escribe haciendo uso de una sintaxis clara y limpia que te permite centrarte en el desarrollo del código. Además, dispone de muchas funciones y librerías que podrás aprovechar, sin necesidad de construirlas desde cero.

- Es un lenguaje interpretado y multiplataforma. El intérprete traducirá tu código sin necesidad de compilarlo, para que el ordenador lo pueda ejecutar a medida que sea necesario y, típicamente, instrucción a instrucción. Así que, haciendo uso de un intérprete, el mismo fichero, es decir, tu código, podrá ser ejecutado en sistemas muy diferentes. Por ello decimos que también es un lenguaje multiplataforma.

Para ser miembro relevante de Python y escribir código de calidad, tendrás que tener en cuenta estos principios, que constituyen *The Zen of Python.*

- Bonito es mejor que feo.
- Simple es mejor que complejo.
- Es importante que se lea bien.
- Disperso es mejor que denso.
- Los errores nunca se deben silenciar.
- Frente la ambigüedad, rehúsa adivinar.
- Ahora es mejor que nunca… aunque nunca es normalmente mejor que ahora mismo.
- Si la implementación es difícil de explicar, será una mala idea.

Quizás estos principios no te digan nada ahora, pero guardarlos en un lugar seguro. Cuando empieces a escribir código, vuélvelos a leer. Te guiarán y te inspirarán.

Escribir código con Python es muy divertido. En este curso, rápidamente empezarás a escribir tus propios algoritmos. Programar te enfrentará a retos emocionantes y te animará a superarte, a crear cosas nuevas a encontrar nuevas soluciones.

Aquí empieza tu camino. ¡Adelante!

1.2 INSTALACIÓN Y FUNCIONAMIENTO DEL INTÉRPRETE

Python es un lenguaje que necesita un intérprete para que el ordenador entienda y ejecute las instrucciones del código. Así que, lo primero que necesitaremos es instalarlo. A continuación, se explican los pasos para hacerlo.

Para ordenadores Mac:

Es posible que tengas instalado, por defecto, un intérprete Python. Puedes comprobarlo en el terminal (recuerda que puedes acceder a él rápidamente utilizando el atajo Cmd + espacio + terminal). Ahí, en el terminal, solo es necesario que escribas *python*. Tanto si no lo tienes instalado (en este caso, te aparecerá un mensaje diciendo que no se encuentra), como si tienes una de las versiones 2.7, es aconsejable instalar el nuevo intérprete que se recomienda a continuación, pues Python 2 dejó de tener mantenimiento en enero del 2020.

Ve a https://www.python.org/downloads/mac-osx y haz clic en la versión más reciente de Python 3, la cual aparece justo debajo de *Python Releases for Mac OS X* (importante: asegúrate de no hacer clic en la versión de Python 2). Cuando estés en la página nueva, ve a la tabla de debajo y selecciona el fichero que se adapte a la versión de tu sistema operativo, que será uno de estos dos:

macOS 64-bit/32-bit installer	Mac OS X	for Mac OS X 10.6 and later	5a95572715e0d600de28d6232c656954	34479513	SIG
macOS 64-bit installer	Mac OS X	for OS X 10.9 and later	4ca0e30f48be690bfe80111daee9509a	27839889	SIG

Importante: si se dispone de la versión del sistema operativo 10.9 (Mavericks) o superior, se recomienda descargar el instalador maOs 64-bit. Guarda el fichero, ve a la sección de

descargas del navegador y haz clic en ella. Sigue los pasos que te marca la ventana que aparece.

Para ordenadores Windows:

Si trabajas con Windows también puedes comprobar si tienes un intérprete Python instalado. Para verlo, abre el Command Prompt y escribe *python*. De la misma manera que para usuarios de Mac, si no lo tienes instalado o tienes una versión 2.7, es mejor instalar el intérprete 3.7.

En este caso, tienes que ir a https://www.python.org/downloads/windows y hacer clic en la versión más reciente de Python 3, tal como se indica en el apartado de Mac. Los pasos son los mismos hasta aquí. Después, una vez estés en la tabla de debajo, puedes descargar uno de estos cuatro ficheros para iniciar la instalación.

Windows x86 executable installer	Windows		ebf1644cdc1eeeebacc92afa949cfc01	25424128	SIG
Windows x86 web-based installer	Windows		d3944e218a45d982f0abcd93b151273a	1324632	SIG
Windows x86-64 executable installer	Windows	for AMD64/EM64T/x64	a2b79563476e9aa47f11899a53349383	26190920	SIG
Windows x86-64 web-based installer	Windows	for AMD64/EM64T/x64	047d19d2569c963b8253a9b2e52395ef	1362888	SIG

Para Windows hay más opciones, ya que se hace diferencia entre aquellos ordenadores que implementan la estructura Intel 64, más formalmente conocida como x86-64, y los que no. Además, ahora también hay la opción de poder descargar un instalador *web-based*, que descargará los componentes que hagan falta durante la instalación.

Una vez lo tengas instalado, vuelve a abrir el terminal (en Mac) o Command Prompt (en Windows) y vuelve a escribir *python*. Si Python 2 ya está instalado en tu ordenador, la forma de ejecutar la nueva versión de Python 3 es con el mandato python3 en lugar de python. Comprueba que, efectivamente, la nueva versión instalada corresponde a la más reciente y escribe *import this*. Tendrás que ver en pantalla como aparecen los lemas que forman *The Zen of Python*. También puedes hacer una pequeña operación. Escribe, por ejemplo, *5 * 3*. ¿Te aparece *15*? Pues ya has usado Python como calculadora; así de rápido y sencillo.

En el caso de que tengas problemas para instalar el intérprete o no te funcione correctamente y no quieras perder demasiado tiempo para intentar encontrar dónde está el error, puedes hacer uso de Python sin necesidad de instalar nada. Para hacerlo, dirigente a la página web: http://www.repl.it/. Aquí, pondrás trabajar de manera totalmente gratuita y seguir todas las lecciones del curso; solo tienes que registrarte con una dirección de correo. Esta plataforma, además, te permitirá acceder a tus ejercicios des de cualquier sitio u ordenador, a través de la nube (*Cloud Computing*).

Después, tendrás que seleccionar Python como tu lenguaje de programación. Si ya utilizas otro lenguaje de programación como Java, Swift u otros, también puedes utilizar esta plataforma para escribir código.

1.3 INSTALACIÓN DEL INTÉRPRETE

¡Hola a todo el mundo!

Seguro que ya tenéis muchas ganas de empezar a programar con Python. Pero antes de eso necesitaremos instalar el intérprete. Este vídeo muestra los pasos de la instalación para un ordenador Mac, aunque los usuarios y usuarias de Windows también lo podrán seguir, ya que los pasos a seguir son prácticamente los mismos.

Antes de nada, nos dirigimos a https://www.python.org/ y hacemos clic en *downloads*. Aquí vemos que se muestran las opciones para instalar el intérprete con Mac o con Windows, y también con Linux/Unix. En este caso, hacemos clic en *Latest Python 3 release* y nos dirigimos directamente a la tabla de debajo de la nueva página, donde encontramos, no solo las opciones de descarga para Mac, sino también para Windows.

Clicamos en este fichero (macOS 64-bit installer), ya que la versión del sistema operativo de este ordenador se adapta a los requisitos correspondientes. Recuerda solo que, si tienes Windows, debes seleccionar el fichero que más te convenga. Para Mac, con una versión inferior a la 10.9, necesitarás instalar el de 32-bit.

Hacemos clic en el fichero y nos aparece la ventana para empezar a descargar el instalador de la ventana de bienvenida. Clicamos en "Continuar", nos aparece la sección *Read Me*, clicamos de nuevo a "Continuar" y así llegamos al apartado de la licencia. Hacemos clic en "Continuar" una vez más. Aceptamos la licencia de uso. Después, seleccionamos a qué disco queremos guardar el intérprete.

Finalmente, tenemos que escoger dónde queremos guardar el fichero dentro del disco, aunque el sistema ya ofrece una opción por defecto para hacerlo. Hacemos clic en instalar y esperamos unos instantes a que se complete la acción. Vemos que la barra está en progreso. Sí, ahora ya está todo listo y podemos cerrar la ventana del instalador.

Ahora solo tenemos que ir a la terminal, para comprobar que el intérprete de Python está instalado correctamente. Si todo hay ido bien, el sistema lo reconoce con un mensaje que indica la versión de Python instalada.

Ya lo tenemos. ¡Hasta pronto!

1.4 INSTALACIÓN Y FUNCIONAMIENTO DEL IDE (ENTORNO DE PROGRAMACIÓN)

Un entorno de programación (también denominado IDE) es un editor de código que permite hacer uso de todas las funcionalidades que ofrece un lenguaje de programación de una manera más ágil, visual, eficaz y ordenada.

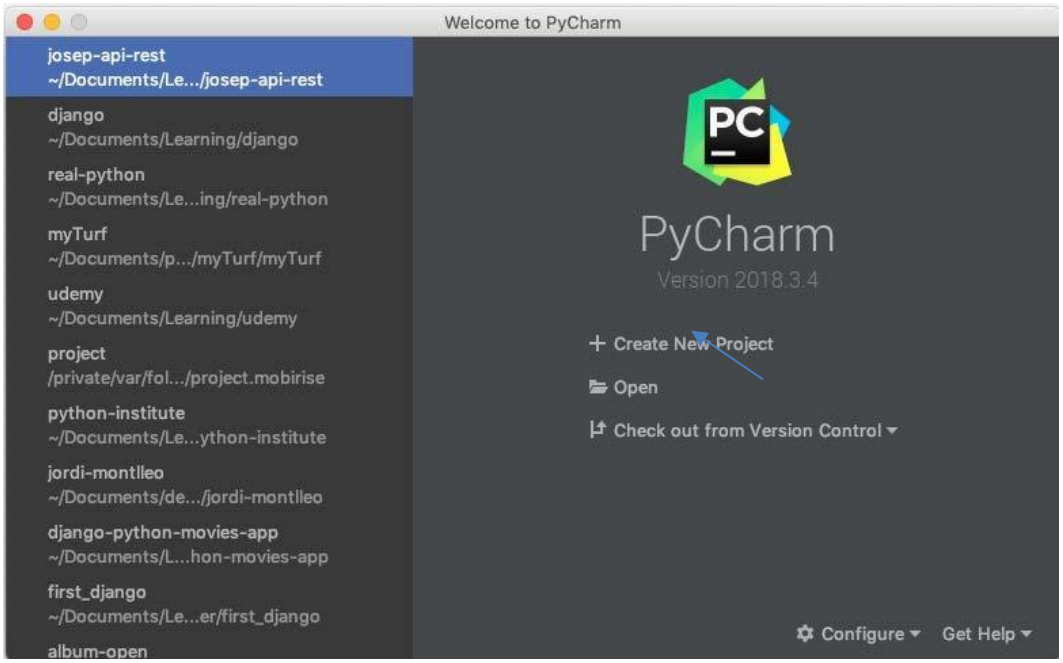
Hay editores de código que pueden ser más específicos para Python, como PyCharm, Spyder o Thonny, y otros que no lo son, pero que también permiten hacer uso de Python, como por ejemplo Eclipse, Sublime Text, Atom, GNU Emacs, Vi/Vim o Visual Studio.

Durante los ejercicios del curso, utilizaremos PyCharm para editar el código hacer correr el fichero. En cualquier caso, si ya tienes experiencia con otros editores que soporten Python, puedes usarlos en vez de PyCharm. Quien trabaja en programación, cuando se enamora de un editor, es difícil que lo cambie por otro. Son muchas horas trabajando y se crea un vínculo. ¡Los programadores y programadoras somos así!

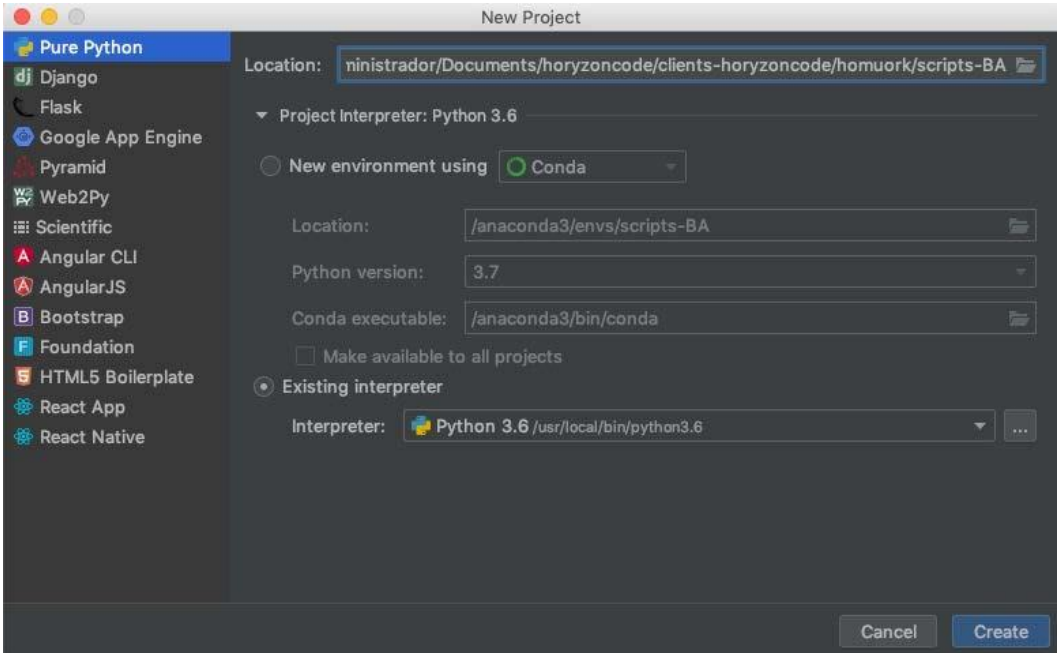
Pasemos, entonces, a instalar la versión gratuita de PyCharm. Primero, solo hace falta que nos dirijamos a la web del proveedor: https://www.jetbrains.com/pycharm/. Ahí seleccionamos qué versión queremos descargar, dependiendo del sistema operativo y de si queremos la versión Professional o la Community. Esta segunda es gratuita. A continuación, nos tiene que aparecer una ventaja para descargar el fichero del instalador del programa. Una vez completada la descarga, seguiremos los pasos para instalar finalmente el programa.

Importante: recordemos que, si queremos ahorrarnos la instalación (o tenemos problemas de compatibilidad), podemos programar con Python sin necesidad de tener el intérprete ni el IDE. Lo podemos hacer directamente desde la red, a través de http://www.repl.it, una aplicación web gratuita.

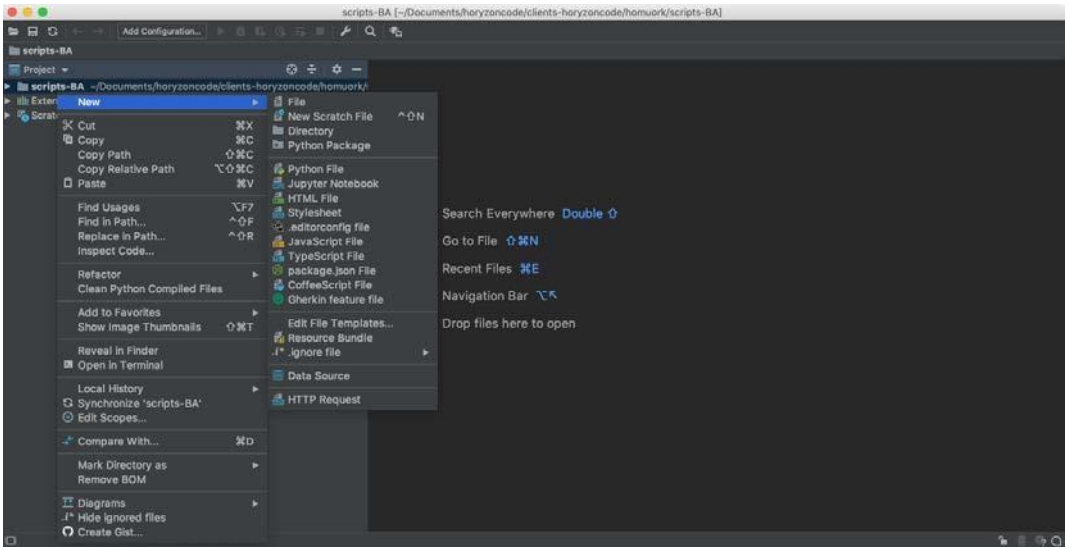
El siguiente paso será abrir PyCharm y crear un proyecto nuevo en el cual guardemos todos los ejercicios que vayamos haciendo durante el curso.



A continuación, seleccionamos dónde queremos guardar el proyecto desde *Location*, marcamos la opción de *Existing Intepreter*, donde tendría que aparecer la versión del intérprete instalado, y hacemos clic en *Create*.



Con el proyecto abierto, iremos al menú vertical de la izquierda. Haciendo clic con el botón derecho, se abrirá la carpeta del proyecto. Así podremos crear un fichero nuevo desde donde escribiremos nuestro código *(New → Python File).*



PyCharm tiene muchas funciones y quizás asusta un poco la primera vez pero, para empezar a programar, solo hace falta aprender un par de herramientas. El programa lo utilizaremos, principalmente, como editor puro, para escribir código y crear las entraras e imprimir los resultados a través de la consola. De esta forma, solo nos tendremos que centrar en aprender el lenguaje Python.

Para comprobar que todo funciona correctamente, escribiremos el código siguiente dentro de la caja de texto, tal y como aparece en la imagen de debajo. Para obtener un resultado, solo tendremos que hacer clic en *play* (para ejecutar la operación *Run*) ahí donde marca el círculo.

print('Te damos la bienvenida a Barcelona Activa')

Lo que hace la operación *Run* es convertir el código escrito en la caja de texto del editor en lenguaje máquina para que el ordenador entienda las instrucciones. En la parte de debajo hay otra caja de texto que muestra los resultados a través de la consola, una vez el ordenador ha ejecutado las órdenes que se le han dado. Cada vez que cambiamos el código dentro del editor y hacemos clic en *Run*, la consola muestra un resultado nuevo, siguiendo las instrucciones nuevas.

Hasta ahora, hemos utilizado una operación muy importante denominada *print*, propia del lenguaje Python, la cual es la encargada de hacer salir por la consola (o pantalla) el contenido entre paréntesis.

Apenas acabamos de empezar. Tu primer algoritmo está cada vez más cerca.

1.5 INSTALACIÓN DEL IDE

¡Hola!

Para poder aprovechar todas las funcionalidades del lenguaje Python de una manera más ágil y completa, necesitaremos un entorno de programación que nos facilite esta tarea.

Encontramos distintas opciones. Unas son más específicas de Python y otras son más generalistas. Para este curso, escogeremos PyCharm como nuestro entorno de programación. Si ya trabajas con otro IDE (recuerda que IDE hace referencia al concepto de entorno de programación) puedes utilizarlo, siempre y cuando este IDE soporte el lenguaje Python. No hace falta que cambies tu manera de trabajar.

Dicho esto, nos dirigimos a la web desde donde podremos descargar PyCharm (https://www.jetbrains.com/pycharm/). La instalación es muy sencilla. Escogemos si queremos la versión de Windows, Mac o Linux, y nos aparece una ventana emergente justo aquí. Guardamos el fichero y esperamos unos instantes a que la descarga finalice. Una vez tengas descargado el fichero del instalador, lo abres y, si eres usuario de Mac, solo hace falta que lo arrastres hasta la carpeta de aplicaciones.

Ahora toca abrir PyCharm. Para empezar, creamos un proyecto nuevo. Podemos escoger el nombre más significativo para nosotros, como por ejemplo la temática que abordará, y seleccionar la ubicación. Una vez creado nos pide qué intérprete queremos utilizar. Seleccionamos *Existing Interpret* y ya tenemos nuestro entorno listo para empezar a trabajar.

No sufras por todas las herramientas y funciones que presenta PyCharm. Solo tendrás que aprenderte un par de ellas. De momento, ve aquí, arriba a la izquierda, donde verás una carpeta con el nombre que has dado al proyecto. Haz clic encima con el botón derecho y selecciona "New Python File". Ya tenemos un fichero listo para empezar a picar código.

¡Empezamos! En esta caja de texto escribimos esta línea de código: *print('Te damos la bienvenida a Barcelona Activa')*. Para comprobar que todo funciona correctamente, solo queda ejecutarla, haciendo clic en *play*. Y, sí, aquí a la consola obtenemos el resultado que queríamos, que era hacer salir por pantalla el texto que hay dentro del paréntesis *print*.

¿Cómo lo ves? Ya has ejecutado tu primer programa de Python y te has comunicado con el ordenador. Y solo es el principio. ¡Seguimos!

1.6 IDEAS CLAVE: QUÉ ES PYTHON Y CÓMO EMPEZAR A PROGRAMAR

Python es uno de los lenguajes de programación que está ganando popularidad en todo el mundo. Ofrece un gran abanico de aplicaciones, entre las que destacan:

- Desarrollo web
- Creación de videojuegos
- Análisis de datos y *big data*
- *Machine learning* e inteligencia artificial
- Automatización de tareas

Las características principales que describen Python son:

- Es un lenguaje de código libre.
- Está basado en la orientación de objetos.
- Es fácil de aprender.
- Es un lenguaje interpretado y multiplataforma.
- Se anima a los programadores y programadoras a diseñar su código siguiendo los lemas de *The Zen of Python.*

El intérprete es el encargado de traducir tu código a lenguaje máquina para que el ordenador ejecute tus instrucciones. Este intérprete es de fácil instalación y solo hace falta seguir las instrucciones para su descarga, desde la web https://www.python.org/downloads/.

Para poder aprovechar todas las funcionalidades de Python de una manera más ágil y ordenada, es conveniente hacer uso de un entorno de programación, también denominado IDE. En nuestro caso, se ha escogido PyCharm. Para poder acceder a la descarga, solo es necesario dirigirse a la web https://www.jetbrains.com/pycharm/ e iniciar los pasos hasta finalizar el proceso.

En el caso de que se quisiera empezar a programar sin necesidad de descargar el intérprete ni el IDE, se puede escribir el código y ejecutarlo des de internet, a través de la aplicación gratuita https://www.repl.it.

Con el fin de empezar a ejecutar código, en la ventana del editor de texto escribiremos nuestro primer código: *print('Aquí iría una frase')* y lo haremos correr para que el intérprete

traduzca nuestras instrucciones a lenguaje máquina. El resultado se imprimirá para pantalla en la ventana de la consola.

2 LAS VARIABLES

¡Hola a todo el mundo!

Ha llegado el momento de hacer uso de las variables con Python. Primero, veremos cómo escribirlas correctamente y de qué se componen, qué tipo de información almacenan y qué acciones podemos utilizar para operar con sus datos o contenido.

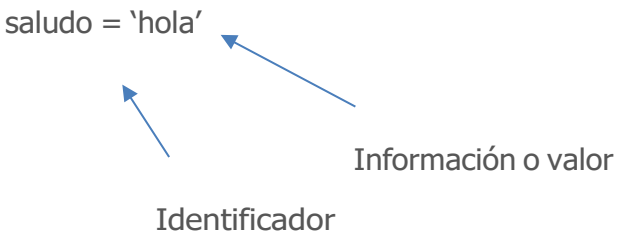
Las variables son las entidades que permiten almacenar la información de los programas. Son, principalmente, espacios de memoria a los que podemos acceder para hacer uso de los datos que guardamos. Aunque en este curso no nos detendremos a analizar la gestión de la memoria, sino que nos centraremos en saber trabajar con las variables desde un punto de vista práctico y muy orientado a la redacción de código.

La sintaxis de las variables con Python es muy sencilla y limpia. Empezaremos por gestionar los tipos de variables más básicos y, luego, veremos algunas variables menos intuitivas que, quizás, requerirán más atención si nunca antes hemos aprendido ningún leguaje de programación.

2.1 LAS DIFERENTES VARIABLES Y SUS OPERACIONES

En este artículo exploraremos qué son las variables, describiremos sus características y explicaremos las operaciones que podemos realizar con ellas. Básicamente, las variables sirven para asignar un espacio de memoria al ordenador para preservar una información o, mejor dicho, para preservar el contenido de un objeto. A Python, cualquier entidad es un objeto.

Así pues, una variable respecto al código estará formada por su identificador (nombre asignado) y el valor o dato de la información (objeto) que contenga. Ejemplo:



A continuación, expondremos los diferentes tipos de objetos más habituales que puede guardar una variable. Son:

Cadena de caracteres (*strings*)

Aquí encontramos las palabras, los textos, direcciones de correo o páginas web (URL), o cualquier otra información que contenga una cadena de caracteres. De hecho, cualquier cadena de caracteres que haga referencia a una información y que esté contenida entre comillas simples o dobles será entendida por Python como una *string*.

Ejemplos:

```
ciudad_nacimiento = 'Barcelona'
lengua = 'Català'
clave_acceso = "htrn!5647@#GF[[}}**"
url = 'http://www.barcelonaactiva.cat'
```

- Números enteros (*int*)

Son aquellos números positivos y negativos que no tienen decimales. Se incluye también el cero.

Ejemplo:

```
edad = 21
errores = 0
```

- Números decimales (*float*)

Son aquellos que tienen decimales, ya sean positivos o negativos.

Ejemplo:

```
puntuacion_media = 3.50
```

- Verdadero o falso (*boolean*)

Este tipo de información solo puede tener el valor de verdad (*True*) o falso (*False*).

Ejemplo:

```
inscripcion_realizada = True
pago_efectuado = False
```

- Otra información como pueden ser las funciones.

Ejemplo:

```
primera_accion = abrir
segunda_accion = cerrar
```

Una nueva variable puede contener el valor de la información de otra. Así que, por ejemplo, podemos escribir lo siguiente:

```
municipio_actual = ciudad_nacimiento
```

El valor que contiene la variable *municipio_actual* es 'Barcelona' (una cadena de caracteres o *strings*) y el *peso_minimo* tiene un valor asignado de 3,50 (un número con decimales o *float*).

Nota: A partir de ahora usaremos nuestro editor de código y veremos resultados en pantalla. Solo tienes que saber lo siguiente:

print() – muestra en pantalla el valor que contiene dentro de los paréntesis.

También es importante que sepas que, si quieres añadir comentarios a tu código para hacer aclaraciones al respecto, puedes hacerlo. Solo tienes que empezar una línea con el símbolo # o escribir el texto entre "'"…"'". Estos comentarios no serán ejecutados por el intérprete y nos servirán para que nuestros compañeros y compañeras que trabajan con nosotros en el mismo proyecto puedan seguir más fácilmente nuestro desarrollo. Los programadores y programadoras trabajan normalmente en equipo. Es así como se pueden llegar a crear grandes proyectos.

Una vez definido el tipo de información u objetos con los cuales trabajaremos y les hemos asignado un espacio en la memoria mediante una variable, veamos qué tipo de operaciones permiten hacer.

- Enteros (*int*) y números decimales (*float*)
  Los números permiten hacer cualquiera de las operaciones matemáticas de sus grupos. Las más básicas son la suma (+), la resta (-), la multiplicación (*), la división (/), etc.

Ejemplo:

```
"""calculamos la nota media del alumnado"""
joan = 8.3
carla = 9.2
adria = 7.5
dolors = 6.9
suma_notas = joan + carla + adria + dolors
n_alumnos = 4
media = suma_notas / n_alumnos
print(media)
```

Importante: Las operaciones siguen el orden de jerarquía convencional.

https://ca.wikibooks.org/wiki/Matem%C3%A0tiques_(nivell_ESO)/Jerarquia_de_les_operacions

En Python, los decimales se escriben siempre con un punto y con con una coma.

Sobre las divisiones, cabe destacar que, si dividimos dos números enteros, el resultado siempre será reconocido por Python como un número de tipo *float*, aunque la división de un resultado entero.

Ejemplo:

```
alumnado = 10
grupos = 2
alumnado_por_grupo = alumnado / grupos
print(alumnado_por_grupo)
```

El resultado, aunque es un 5 redondo, Pyhon lo reconoce como 5.0.

- Cadena de caracteres:

Las cadenas de caracteres también permiten la suma y la multiplicación. Ejemplo:

```
# sumamos dos palabras y después las multiplicamos por 2
p1 = 'hola'
p2 = 'barcelona'
saludo = p1 + ' ' + p2
print(saludo)
print((saludo,) * 2)
```

Verás que un espacio también puede ser representado como carácter. Con la función *print()* se puede imprimir una secuencia de *strings* separada por comillas. Estas comas representan un espacio en la pantalla.

Como acabas de ver, la función *print()* acepta operaciones dentro de sus paréntesis. Eso quiere decir que, primero, resuelve la operación y, luego, imprime el resultado.

Con eso ya puedes empezar a escribir código. Quédate con nosotros y aprenderás aún mucho más.

2.2 PARTICULARIDADES DE LAS VARIABLES

¡Hola a todo el mundo!

Seguimos con las variables y sus operaciones. Ahora veremos cómo entrar por teclado una información, para así guardarla en una variable.

Escribimos este pequeño código:

```
saludo = 'Hola'
print('tu nombre:')
nombre = input()
print(saludo, nom)
```

Sí, crear entradas por teclado es así de sencillo. Solo debemos tener en cuenta que las entradas desde *input()* siempre son gestionadas como cadenas de caracteres (*strings*). Es como si nosotros siempre entráramos la información entre comillas. Por ejemplo:

```
n1 = 5
print('otro numero:')
n2  =  input()
print(n1 + n2)
```

Por teclado, introducimos 5 y… ¡hala!, ya tenemos un error. Pero es normal, ya que hemos intentado sumar un número entero (int) con una *string* (int), y eso no se puede hacer. Recuerda que todas las entradas (*input()*) serán *strings*. Te estarás preguntando cómo podemos operar con números, ¿verdad?. Pues bien, tenemos mecanismos para modificar el tipo de información con la cual estamos trabajando. Mira, si n2 lo ponemos dentro del

paréntesis de *int()*, Python convertirá el valor introducido, que es del tipo *string*, en número entero. Y ahora sí que podemos hacer la suma.

Bien, ¿no? Cada vez podemos hacer más cosas con Python.

Para que un número entero se convierta en una *string* lo hacemos así. Hemos puesto el valor dentro de los paréntesis de *st()*, de esta manera:

```
x = 8
y = str(x)
```

Y, si lo necesitamos, podemos cambiar un número *float* por un número entero.

```
x = 10
y = 5
z  =  10/5
resultado = int(z)
print(resultado)
```

El resultado que obtnemos no es un 2.0 (un *float*) sino un 2 (un *int*). Obviamente, también podemos pasar de *float* a *string* y viceversa. Recuerda que las divisiones dan siempre valores del tipo *float*.

Ahora revisamos los *boolean* (valores que solo pueden ser verdad o falso). Estos también se pueden convertir en *int* o *float*. Lo hacemos así:

```
respuesta1 = True
respuesta2 = False
print(int(resposta1))
print(float(resposta2))
```

Sí, efectivamente. El valor *True* cuando pasa a ser un número entero es 1 y el valor *False* pasa a 0. Si pasa a *float*, entonces *True* es 1.0 y *False* es 0.0.

Finalmente, si quieres saber exactamente qué tipo de información contiene una variable, solo hace falta que escribas esta declaración:

```
x = 10.5
y = 'pelota'
print(type(x))
print(type(y))
```

Efectivamente, *X* contiene un valor del tipo *float* y *Y* es del tipo *string*.

¡Hasta el próximo vídeo!

2.3 AHORA TE TOCA A TI: RETO. CALCULA EL VALOR DE X

Supongamos que hacemos distintas compras al año, en total cuatro, a una empresa proveedora que nos vende sus productos en dólares. Los importes de las compras son: 356.75 $, 487.45 $, 295.83 $ y 532.00 $. Tendremos que convertir los valores de dólares a euros.

Parte 1: Calcula e imprime en pantalla los datos siguientes:

- Suma total de las compras en euros
- Media de las cuatro compras en euros

Nota: procura que la variable que necesitarás para tener el cambio de dólares a euros reciba el dato a través de una entrada por teclado.

Parte 2: Imagina que el último pedido (el de 532.00 $) ha tenido problemas de género en buenas condiciones y, por tanto, le decimos al proveedor que solo le pagaremos el 80% del importe acordado previamente.

- Calcula ahora los mismos resultados que en el caso anterior e imprímelos en pantalla.
- ¿Podrías decir qué tipo de dato (información) es la suma total de cualquiera de los dos casos?
- ¿Sabrías cambiarla a un dato del tipo *string*?

Procura esforzarte para resolver ambas partes del ejercicio sin necesidad de mirar la solución. Si no loconsigues, entonces consúltala.

¡Adelante!

2.4 SOLUCIÓN EJERCICIO

Antes de empezara a picar código, es importante entender qué nos pide el problema y saber mentalmente qué pasos deberemos seguir para llegar a la solución óptima. En programación, es importante, no solo hacer que tu código funciones, sino que también resuelva el problema de la manera más eficiente posible. Recuerda que puedes repasar los lemas de *The Zen of Python* para orientarte.

En este caso, lo primero de todo es crear las variables y guardar allí los importes de las cuatro compras y el cambio de dólares a euros. Asumiremos que el cambio de dólares a euros es de 0,890.

```
c1 = 356.75
c2 = 487.45
c3 = 295.83
c4 = 532.00
print('Introduce cambio:')
cambio = input() # cambio dolares a euros
cambio = float(cambio)
```

Nota: recuerda que todos los *inputs* que entren como *string* los tenemos que cambiar a *float* para realizar operaciones con las compras.

Acto seguido, podemos calcular la suma de las compras e imprimir el resultado:

```
suma_euros = (c1 + c2 + c3 + c4) * cambio
media = suma_euros / 4
```

```
print('Suma de las compras {:.2f} €'.format(suma_euros))
print('Media de las compras: {:.2f} €'.format(media))
```

Resultado de la suma = 1.488, 11 €

Resultado de la media = 372,03 €

Ahora, resolveremos el segundo caso en el cual solo tendrás que pagar el 80% del importe de la cuarta compra:

```
suma_euros_2 = (c1 + c2 + c3 + c4*0.8) * cambio
media_2 = suma_euros_2 / 4
```

```
print('Nueva suma: {:.2f}'.format(suma_euros_2))
print('Media de las compras: {:.2f} €'.format(media_2))
```

Nota: fíjate en el código del cuadro de arriba. Dentro de los { } se introducirá el valor de la variable que está en *format()* y se escribirá con dos decimales. Esto servirá para reducir los decimales a solo dos.

Resultado de la suma nueva = 1.393,41€

Resultado de la media nueva = 348,35€

Finalmente, para saber qué tipo de dato es la suma total (cualquiera de los dos casos) y pasarla a una *string*, tenemos que hacerlo así:

```
suma_type = type(suma_euros)
suma_string = str(suma_euros)
```

2.5 IDEAS CLAVE: LAS VARIABLES

A continuación, haremos un repaso de las ideas más importantes que hemos tratado.

Qué es realmente una variable.

- Cómo declararla en el código.
- Qué tipos de información pueden guardar las variables.
- Qué operaciones podemos hacer con los tipos de datos.
- Cómo podemos pasar un tipos de dato a otro.

También hemos aprendido a entrar un dato por teclado y guardarla en una variable. A continuación, hemos sido capaces de imprimir resultados en pantalla e, incluso, especificar cuántos decimales queremos mostrar.

Y, no menos importante, ahora sabemos escribir comentarios en nuestro código. Estos comentarios no serán ejecutados por el intérprete y nos servirán para que la gente de nuestro equipo pueda seguir más fácilmente nuestro desarrollo.

3 ESTRUCTURA DE DATOS. LISTAS Y DICCIONARIOS

Ha llegado el momento de trabajar con estas dos tipos de estructuras de datos: las listas y los diccionarios. También profundizaremos sobre las acciones que podemos hacer con ellas, como añadir o retirar elementos, acceder a la información, etc.

Las listas son las entidades (objetos) que permiten guardar una colección de datos dentro de una misma variable. Los diccionarios también, pero en este caso necesitaremos una clave para acceder a la información.

Durante la redacción del código es muy habitual hacer uso de estas estructuras, así que este módulo es muy importante para que, una vez hayamos entendido cómo trabajar con ellas, hagamos un salto significativo como futuros programadores y programadoras.

3.1 TRABAJAR CON LISTAS

Las listas son una colección de datos (objetos) que pueden ser del mismo tipo o no. Estas listas, entonces, permiten almacenar cualquier tipo de datos con Python: *int*, *float*, *string*, *boolean* y cualquier otro objeto. Recuerda, en Python todo son objetos.

Las listas son almacenadas en la memoria, cuando son asignadas a una variable. A continuación, se expone una lista con una colección de números enteros (*int*), la cual es asignada a una variable denominada *lista_enteros*.

```
lista_enteros = [4,6,3,7,8]
```

Las listas exponen sus variables separadas por comas y la colección entera tiene que estar dentro de dos corchetes: uno al inicio y otro al final. Aquí se muestra otro ejemplo. En este caso, la misma lista incluye datos de diferentes tipos.

```
lista_mixta = [-4, 7.8, 'hola', False]
```

Cada dato de dentro de una lista tiene asignada una posición. La primera posición es la que está más a la izquierda. Su índice es 0. La segunda posición es 1, después 2, y así sucesivamente hasta el último dato que se encuentra más a la derecha. Para poder acceder a un dato de la colección y guardarlo en una variable, lo hacemos así:

Importante: la primera posición siempre es 0 y no 1.

```
a = lista_enteros[0]
b = lista_mixta [3]
```

Fíjate que el valor de *a* es 4 y el valor de *b* es *False*. Si quieres imprimir los datos de una colección sin necesidad de crear una variable, lo puedes hacer directamente de la manera siguiente:

```
print(lista_mixta[2])
```

El resultado que muestra la pantalla es *hola*.

Esta selección también se puede hacer indicado el índice correspondiente empezando desde la derecha. En este caso, se utilizarán números negativos. La última posición tiene asignado el índice -1. A partir de aquí, hacia la izquierda, los índices son -2, -3, etc.

Por ejemplo, si queremos extraer de *lista_mixta* el dato *7.8*, lo podemos hacer de estas dos formas:

```
a = lista_mixta[1]
b = lista_mixta[-3]
```

Si quieres saber la cantidad de datos que contiene una lista, lo puedes hacer llamando a la función *len()*, tal como se ve en el ejemplo:

```
print(len(llista_enteros))
```

Este código permite imprimir en pantalla la cantidad de objetos que contiene la *lista_enteros*, que tiene has cinco valores.

Acto seguido, exponemos las acciones con las que podemos gestionar datos de una lista.

- Añadir un nuevo dato – *append()*:

```
llista_mixta.append(True)
```

La sintaxis de esta acción se compone de la manera siguiente: primero, se escribe el nombre de la variable que hacer referencia a la lista en cuestión; acto seguido se coloca un punto y, justo a la derecha, se escribe la acción de añadir. *Append* indica la acción de añadir el dato que está dentro del paréntesis.

Importante: el dato u objeto añadido a una lista mediante *append()* siempre se colocará en la última posición (la que está más a la derecha).

- Extraer o eliminar el último dato – *pop()*:

Para eliminar el último dato añadido, llamaremos a la función *pop()*, siguiendo la misma sintaxis con con la acción *append()* aunque, para esta nueva función, no hace falta añadir nada en los paréntesis.

Pongamos un ejemplo:

```
lista_enteros.pop()
print(llista_enters)
```

Si ahora imprimes la *lista_enteros* para ver qué datos muestra, verás que la última posición ha sido eliminada.

- Identificar la posición de un dato – *index():*

Si quieres saber la posición que ocupa un dato dentro de la colección, tendrás que ejecutar esta acción, siguiendo el mismo modelo que las otras dos anteriores:

```
print(lista_mixta.index('hola'))
```

Como podemos ver, dentro del paréntesis tiene que ir el dato del cual queremos aber la posición que ocupa en la lista.

- Hacer una copia de una lista, seleccionando solo ciertas posiciones:

Si quieres utilizar solo una parte de una colección y guardarla en otra variable, tendrás que seguir el método siguiente:

```
a = lista_enteros[0:3]
```

La variable *a* contiene una lista que es una copia de *lista_enteros* desde la posición 0 hasta la posición 2. El índice de la derecha no se incluye. Podemos poner otro ejemplo:

```
notas_alumnos = [8.75, 3.50, 5.50, 6.25, 9.75, 2.75, 4.50, 7.60]
notas_ultimas = notas_alumnos[3:]
```

notas_ultimas tendría los valores: 6.25, 9.75, 2.75, 4.50 y 7.60.

Si el índice de la derecha lo dejamos en blanco, el intérprete entenderá que tiene que coger todas las variables desde la posición de la izquierda hasta el final. Si, en cambio, dejamos en blanco el índice de la izquierda, la copia empezará desde la posición 0. Veamos este último ejemplo:

```
notas_ultimas = notas_alumnos[:5]
```

Ahora *notas_utlimas* tendría estos valores: 8.75, 3.50, 5.50, 6.25 y 9.75.

Recuerda que el índice de la derecha no está incluido y que, por tanto, la última posición es la que se encuentra antes.

3.2 TRABAJAR CON DICCIONARIOS

Los diccionarios son una colección de datos que permiten acceder a la información que contienen mediante una clave. Este tipo de estructuras de datos también se conocen por clave-valor. A continuación, se muestra un ejemplo para ver la estructura y cómo se puede escribir un diccionario.

```
notas_dict = {'Joan': 4.85, 'Miquel': 7.80, 'Meritxell': 8.15, 'Laura': 4.75}
```

También podemos crear diccionarios con el método dict. Mostramos un ejemplo:

```
notas_dict = dict (Joan= 4.85, Miquel= 7.80, Meritxell= 8.15, Laura= 4.75)
```

Aquí tenemos, pues, el nombre de la variable *notas_dict* donde se guarda e diccionario. Las claves son 'Joan', 'Miquel', 'Meritxell' y 'Laura'. Los valores son 4.85, 7.80, 8.15 i 4.75. También vemos que, para abrir y cerrar un diccionario, se utilizan los caracteres { }. Para las listas, en cambio, recuerda que se utilizan corchetes [ ].

Si queremos acceder, por ejemplo, a la nota de Meritxell y guardarla en una variable, lo haremos así:

```
nota_meritxell = notes_dict['Meritxell']
```

Importante: los diccionarios aceptan cualquier tipo de dato como valor. Para las claves, normalmente se utilizan *strings* o números enteros.

También podemos acceder al valor de una clave determinada haciendo uso de la función *get()*. Se haría así:

```
notas_dict.get('Laura')
```

Esta línea de código nos permite acceder al valor 4.75.

Si queremos añadir una nueva parella clave-valor, solo tendremos que escribir esta declaración:

```
notas_dict['Eduard'] = 6.80
```

Ahora, el diccionario tiene una nueva clave denominada 'Eduard' con un valor de 6.80.

Para saber el número de claves-valor que contiene un diccionario, lo podemos hacer de la misma manera que con las listas: tenemos que llamar a la función *len()* y poner dentro del paréntesis el nombre de la variable que contiene el diccionario:

```
print(len(notas_dict))
```

En este caso, nos aparece en pantalla el número de registros clave-valor para porque hemos llamado a la función *len()* dentro de la función *print()*.

Si queremos, por ejemplo, extraer la colección de claves en forma de lista, lo podemos hacer con dos pasos:

```
claves = notas_dict.keys()
claves_lista = list(claves)
```

La función *keys()* está asociada a los diccionarios. Esta función genera una colección que contiene las claves del diccionario, la cual se puede convertir en una lista haciendo uso de la función *list()*.

Aquí estamos haciendo un cambio de tipo de objeto de la misma forma que lo hemos hecho antes entre los tipo *int* y *float*.

También se puede generar una lista con la colección de valores. Lo haremos de la misma manera que con las claves, pero en lugar de llamarlo con la función *keys()* llamaremos la función *values()*.

```
valores = notas_dict.values()
valores_lista = list(valores)
```

Esto nos puede servir, por ejemplo, para saber si una clave se encuentra en un diccionario o no. Para saber si está o no, lo haremos así:

```
print('Kim' in claves_lista)
```

Este código nos imprimirá *True* (si está presente) o *False* (si no está presente).

Y, si por alguna razón nos interesa generar una lista con las parejas clave-valor, lo podemos hacer llamando a la función *ítems()*. El código se escribirá así:

```
parejas = notas_dict.items()
parejas_lista = list(parejas)
```

Para eliminar una pareja clave-valor del diccionario, lo podemos hacer a través de esta declaración:

```
del notas_dict['Joan']
```

Ahora, la pareja clave-valor compuesta por 'Joan' y 4.85 ha quedado eliminada.

Un diccionario puede tener una clave cuyo valor sea otro diccionario. Recuerda que los valores pueden ser cualquier tipo de objeto y los diccionarios también son un tipo de objeto. Podemos escribir este ejemplo para verlo más claramente:

```
nested_dict = {'a': 1, 'b': 2, 'c': {'c1': 3, 'c2': 4}, 'd': 5}
```

Una lista también puede ser un valor de una clave de un diccionario. Este sería un ejemplo:

```
nested_dict = {'a': 1, 'b': 2, 'c': [3,4,5,6], 'd': 7}
```

Si queremos acceder e imprimir el valor de la posición 2 de la lista, que es el valor de la clave 'c', se podría hacer así:

```
print(nested_dict['c'][2])
```

Primero, accedemos a lista mediante la clave 'c' y, luego, accedemos a la posición de la lista que nos interés escribiendo el índice que corresponda.

Los diccionarios son muy útiles y aún nos quedan muchas cosas por descubrir de Python.

¡Seguimos!

3.3 JUGAR CON DATOS

En este vídeo haremos un ejercicio sobre diccionarios y listas que te servirá para después poner en práctica los conocimientos de manera autónoma.

¡Empezamos!

Una vez tienes PyCharm abierto, puedes crear otro fichero de Python. Recuerda, botón derecho encima de la carpeta, selecciona *new* y después *Python File*.

Supongamos que ha llegado el verano y quieres saber cuántas camisetas tienes en el armario en función del color. Para ello creas un diccionario. Pondremos el ejemplo siguiente:

```
camisetas = {'rojas': 1, 'blancas'; 5, 'azules': 2, 'negras': 3}
```

Ves como tenemos las claves del diccionario y sus respectivos valores, ¿verdad? Sí, los colores en forma de *strings* son las claves y los números (en este caso, enteros) son los valores de cada clave. Hasta aquí bien, ¿no?

Imagina que ha pasado un tiempo y no encuentras una camiseta de color amarillo. Así que vas al diccionario de camisetas y compruebas si tenías alguna de color amarillo. ¿Cómo lo hacemos para saberlo?

Pues debemos conocer si alguna de las claves corresponde a 'amarillas'. Primero, entonces, generamos la colección de las claves y después transformamos esa colección en lista.

```
claves = camisetas.keys()
print('amarillas' in list(claves))
```

Haz memoria, si se nos imprime el valor *boolean False*, querrá decir que no tenemos camisetas amarillas. En cambio, si cambiamos 'amarillas' por 'rojas', veremos que se imprime *True* y, por tanto, querrá decir que sí tenemos camisetas rojas en el armario.

Imagínate que, después de ver que no tienes ninguna camiseta amarilla en el armario, te propones comprarte una y añadirla al diccionario. ¿Recuerdas cómo hacerlo?

Solo tenemos que escribir esto:

```
camisetas['amarillas'] = 1
```

Si te compras dos, escribirás 2 en lugar de 1.

Comprobamos ahora que, efectivamente, tenemos 'amarillas' incluido a la lista que contiene las claves del diccionario:

```
claves = camisetas.keys()
print('amarillas' in list(claves))
```

Fíjate que hemos tenido que volver al mismo código otra vez. Cuando domines la parte de las funciones, no te hará falta hacerlo, pero poco a poco. ¡Todo llega!

¿Ves que en la primera comprobación se imprime *False*, pero en la segunda obtenemos *True*?

¿Comprobamos qué posición tienen 'amarillas' en la lista de las claves? Sí, venga, hagámoslo.

```
print(list(claves).index('amarillas'))
```

¡Posición 4!

Vamos a sumar el número de camisetas (valores) sin necesidade de hacerlo manualmente de una en una. Las listas permiten sumar sus valores en un solo paso:

```
valores_lista = list(camisetas.values())
print(sum(valores_lista))
```

Sí, la función *sum()* es la que nos permite hacerlo. Así de fácil.

Ahora pruébalo tú. Seguro que lo consigues.

¡Hasta pronto!

3.4 AHORA TE TOCA A TI: RETO. LA CESTA DE LA COMPRA

Fíjate bien en el enunciado y completa el ejercicio siguiente:

Supongamos que hemos realizado una compra en una frutería. Las frutas y su importe nos aparecen en el recibo de la manera siguiente:

- Manzanas: 3,56 €
- Mandarinas: 4,35 €
- Sandía: 6,23 €
- Fresas: 4,28 €
- Peras: 2,86
- Naranjas: 3,48 €

Guarda la lista de la compra en forma de diccionario (puedes omitir las unidades). Escribe el código para resolver las cuestiones siguientes:

- ¿Podrías calcular la media de la compra accediendo a los valores de las claves? Procura escribir solo dos decimales.
- ¿Podrías copiar y guardar en una nueva variable la lista de los importes sin tener en cuenta los dos últimos?
- ¿Podrías saber cómo comprobar si hemos comprado limones?

¡Adelante!

3.5 SOLUCIÓN EJERCICIO

¿Cómo ha ido el reto de la cesta de la compra? A continuación, podemos ver los resultados paso a paso.

Primero, tenemos que construir nuestro diccionario según la lista de productos que tenemos en la lista.

Así que el diccionario se escribiría así:

```
compra = {'manzanas': 3.56, 'mandarinas': 4.35, 'sandia': 6.23,

'fresas': 4.28, 'peras': 2.86, 'naranjas': 3.48}
```

Recuerda, los diccionarios se escriben como una colección de claves-valor y entre { }, a diferencia de las listas que van entre [ ]. Las claves siempre son parte de la pareja que está a la izquierda de los : y los valores están a la derecha.

Este diccionario, tal como se muestra, lo guardamos en una variable denominada *compra*. En este caso, las frutas son *strings* y los valores son *floats*. Es importante escribirlo bien.

La primera cuestión que tenemos para resolver es calcular la media de los importes. La estrategia a seguir sería pasar todos los valores a una lista; después sumarla y dividir la suma por el número de valores contiene la misma lista. Lo hacemos así:

```
valores = list(compra.values())
media = sum(valores)/len(valores)
```

Fíjate que hemos guardado la lista de los valores en una variable que hemos denominado *valores*.

Si imprimimos esta media, vemos el resultado:

```
print('La media de la compra es {:.2f}'.format(media))
```

Aquí se muestra cómo reducir el nombre de decimales a solo 2.

La segunda cuestión es ver cómo copiar la lista de valores descartando los dos últimos datos. La solución a la cuestión se resuelve con estas líneas de código:

```
nuevos_valores = valores[:-2]
print(nuevos_valors)
```

Recuerda que una lista puede generar una copia suya de unos determinados índices, haciendo uso de los claudators [index_inicial: index_final(no incluido)]. Si el índice inicial es omitido, la posición inicial es 0 y, si el índice final se omite, se copiarán todos los valores hasta la posición final incluida.

¿Cómo podemos comprobar si hemos comprado limones? Para hacerlo, recuerda que solo nos hará falta mirar si los limones están en la lista de las claves del diccionario. El primer paso, entonces, es convertir todas las claves en una lista.

```
print('limones' in list(compra.keys()))
```

Fíjate que aquí hemos resuelto la cuestión en una sola línea de código. En los ejercicios anteriores, habíamos necesitado más de una. Con Python, las funciones se pueden ejecutar una dentro de otra. A medida que vayas acumulando experiencia, esta habilidad te será muy útil para escribir código más limpio y más rápido.

Si quieres saber si una fruta que entras por teclado está dentro de la lista, el código solo cambiaría de esta manera:

```
print('Entra la fruta nueva: ')
fruta = input()
print(fruta in list(compra.keys()))
```

En este caso, el intérprete evaluará si el dato que contiene la variable fruta está dentro de la lista. Y, como le hemos asignado a fruta el valor de la entrad por teclado, estamos evaluando aquella fruta que nosotros hemos descrito.

3.6 IDEAS CLAVE: ESTRUCTURA DE DATOS. LISTAS Y DICCIONARIOS

En este módulo hemos aprendido a trabajar con listas y diccionarios, que son las estructuras de datos más utilizadas en Python.

Las listas son una colección de datos, del mismo tipo o no, a los cuales se puede acceder mediante un índice de posición. Cada dato de la lista tiene asignado su propio índice. Hay una particularidad y es que el primer índice siempre es 0 y no 1. Además, con las listas, has aprendido las acciones siguientes:

- Saber si un dato forma parte de la lista mediante la sentencia *in*, la cual devuelve un valor verdadero o falso, dependiendo de si forma parte o no de ella.
- Añadir un nuevo dato con *append()*.
- Quitar el último dato con *pop()*.
- Saber qué posición tiene un dato con *index()*.
- Sumar los valores que contiene una lista con *sum()*.
- Saber el número de datos que contiene una lista con *len()*.

Los diccionarios son una colección de datos formados por parejas clave-valor. Para acceder a la información de un dato, hace falta saber la clave asociada y no su posición, como sucede con las listas. Con los diccionarios, has aprendido a:

- Añadir una nueva pareja clave-valor.
- Saber cuántas parejas clave-valor tiene un diccionario.
- Hacer uso de la función *get()* para acceder a una información.
- Generar una lista del conjunto de claves.
- Generar una lista del conjunto de valores.
- Generar una lista del conjunto de parejas clave-valor
- Eliminar una pareja clave-valor.

También hemos visto que los objetos, las funciones y las estructuras de datos están conectadas entre ellas y no son entidades aisladas.

4 ESTRUCTURAS CONDICIONALES Y CICLOS

En las próximas actividades darás realmente un paso adelante para convertirte en programador o programadora y conseguirás las habilidades que te permitirán diseñar tu primer algoritmo. Veremos las sentencias condicionales y los ciclos.

Las sentencias condicionales son bloques de código que nos sirven para ejecutar una serie de acciones u otras, dependiendo de unas determinadas condiciones. Estas estructuras, pues, hacen posible diseñar algoritmos capaces de resolver problemas muy complejos.

Los ciclos son bloques de código que se repiten siempre que se cumpla una condición. Esta herramienta permite repetir una serie de acciones sin necesidad de repetir las mismas líneas de código.

Las condiciones y los ciclos nos esperan. ¡Empezamos!

4.1 DOMINAR LAS ESTRUCTURAS CONDICIONALES. PRIMER ALGORITMO

Las sentencias condicionales permiten ejecutar un código u otro, dependiendo del cumplimiento, o no, de una condición previa.

Aquí se expone un ejemplo sencillo de cómo se escribe una estructura condicional:

```
n = 8
if n > 10:
    print('mayor que 10')
elif n < 10:
    print('menor que 10')
else:
    print('igual que 10')
```

En este caso, estamos diciendo a la herramienta que, si la variable *n* es mayor que 10, imprima *'mayor que 10'* y, si es menor que 10, imprima *'menor que 10'*; si *n* no es ni menor ni mayor que 10, entonces le decimos que imprima *'igual que 10'*.

Si nos fijamos, podemos traducir las sentencias siguientes de esta manera, partiendo del inglés:

- if n > 10 significaría 'en caso de que *n* sea mayor que 10'
- elif n < 10 significaría 'si no, y en caso de que *n* sea menor que'
- else 'si no…' (es decir, cualquier otra posibilidad)

Las sentencias *if*, *elif* y *else* siempre terminan en : – dos puntos – y el código que se ejecuta en el caso de cumplirse la condición siempre se tendrá que indexar, como mínimo, una posición más hacia la derecha. Esta sintaxis es muy característica de Python. A diferencia de

otros lenguajes de programación, las condiciones no van entre paréntesis y los bloques de código no se tienen que escribir entre '{}'; solo se tienen que indexar.

La sentencia *if* siempre va en primer lugar.

Las sentencias *elif* (una o más de una) van en segundo lugar.

La sentencia *else* siempre va al final.

A continuación, introduciremos los operadores comparativos, que emiten un resultado de verdadero o falso, según si se cumple la condición o no:

> (mayor que)

< (menor que)

>= (mayor o igual que)

<= (menor o igual que)

== (igual que)

!= (no igual que)

Si utilizamos estos operadores dentro de la función *print()*, lo veremos.

```
n = 10

print(10 < n)
```

En este caso, se imprimirá *False*.

```
if n = 10:
```

Nota: El código de arriba dará error. Para evaluar una condición (igual o no) se debe hacer uso del operador ==. Trabajaremos ahora con listas y condicionales juntas:

```
nombres = ['Ernest', 'Laura', 'Miquel', 'Maria']

if nombres[1] == 'Laura':
    print('nombre correcto')
else:
    print('nombre incorrecto')
```

En este caso, estamos comprobando si el nombre de la posición 1 de la lista *nombres* es 'Laura'. En caso afirmativo, se imprime *'nombre correcto'* y, en caso contrario, se ha de imprimir *'nombre incorrecto'*.

¿Recuerdas la sentencia *in* que utilizábamos para ver si un dato estaba dentro de una lista? Pues bien, esta la puedes utilizar junto con las condicionales. Lo haremos de esta forma.

```
nombres = ['Ernest', 'Laura', 'Miquel', 'Maria']

if nombres[1]  in  nombres:
    print(nombres[1],'está presente en nombres')
else:
    print(nombres[1], 'no está presente en nombres')
```

Aquí estamos utilizando la sentencia *in* como condición, el resultado de la cual determinará si se ejecuta un código u otro.

En este caso, como si existe la primera posición en la lista "nombres" dará el resultado de "Ernest está presente en nombres".

Nota: dentro de la función *print()*, las comas sirven para encadenar *strings*, añadiendo un espacio.

Veamos otro ejemplo. En este caso, haciendo uso de números para ver operadores comparativos:

```
if n >= 10:
    print('mayor o igual que 10')
elif n <= 5:
    print('menor o igual que 5')
else:
    print('mayor que 5 y menor que 10')
```

Aquí estamos evaluando si *n* es mayor o igual que 10 o bien si *n* es menor o igual que 5. En el caso de que ninguna de las dos condiciones se cumpla, se imprimirá '*mayor que 5 y menor que 10*'.

Veamos una estructura condicional que de pie a ejecutar un código más complejo.

```
nombre_nuevo = input()
n = int(input())

nombres = ['Ernest', 'Laura', 'Miquel', 'Maria']


if n <= 20:
    nombres.append(nom_nou)
else:
    nombres.pop()

print(nombres)
```

Lo que estamos haciendo aquí es, primero, introducir por teclado un nuevo nombre y un número. Recuerda que todos los *inputs* son *strings* (por eso hacemos uso de la función *int()* para hacer el cambio a número entero). Después, el número introducido lo escribimos como condición y, según si este número es menor o igual que 20, se añadirá a la lista el nuevo nombre o, en el caso contrario, se eliminará el de la última posición.

También podemos utilizar las condicionales y los diccionarios juntos. Aquí tenemos un ejemplo:

```
datos = {'nombres': 'Miquel', 'ciudad': 'Barcelona', 'edad': 28}

if datos['ciudad'] == 'Barcelona':
    datos['barrio'] = 'Eixample'

if datos['edad'] < 30:
    datos['categoria'] = 'Joven'

print(datos)
```

Aquí estamos evaluando dos valores de dos claves diferentes del diccionario. En el caso de que la ciudad sea Barcelona, se añadirá una nueva clave-valor *(barrio: 'Eixample')*. Además, también se evalúa si la edad es menor que 30 y, en el caso de que sea así, el diccionario tendrá otra pareja clave-valor *(categoría: 'Joven')*.

En este caso, se imprimirá: `{'nombre': 'Miquel', 'ciudad': 'Barcelona', 'edad': 28, 'barrio': 'Eixample', 'categoría': 'Joven'}`.

4.2 REPASO DE LAS ESTRUCTURAS CONDICIONALES *IF*

¡Hola!

¿Cómo ha ido la introducción a las estructuras condicionales? Ahora haremos un repaso e iremos, incluso, un poco más lejos. ¡Vamos allá!

Escribiremos un pequeño programa que determine qué acciones ejecutar, según el tiempo que haga. Empezaremos declarando dos variables (temperatura y probabilidad de lluvia), que tendrán los datos que entremos por teclado.

```
temp = int(input())
prob = float(input())
```

Acto seguido, añadiremos un diccionario que indicará sin llevamos chaqueta o no, qué tipo de pantalones y qué tipo de calzado. Finalmente, escribiremos un código que modifique los valores de las claves del diccionario, según la temperatura y probabilidad de lluvia.

```
temp = int(input())
prob = float(input())


dic = {'chaqueta': False, 'pantalones': 'cortos', 'calzado': 'normal'}

if temp <= 20:
```

```
    dic['chaqueta']= True
    dic['pantalones']= 'largos'
    if prob > 75:
        dic['calzado']='invierno']
```

¿Te has fijado? Dentro del código del primer *if*, hemos añadido otro *if*. ¡Realmente no tenemos ningún límite para añadir *ifs* dentro de otros! Básicamente, lo que estamos haciendo es hacer que una condición primero dependa de otra que es previa. En este caso, por ejemplo, la condición de mirar la probabilidad de lluvia se evalúa después de haber evaluado primero la temperatura.

```
elif temp > 20:
    if prob > 80:
        dic['calzado']= 'botas de agua'
```

En el caso de que la temperatura sea superior a 20, solo podremos cambiar el valor de la clave 'calzado', en el caso de que la probabilidad de lluvia sea mayor que 80. Veamos, pues, que nuestro criterio para evaluar la condición de lluvia varía según una condición previa.

```
temp = int(input())
prob = float(input())


dic = {'chaqueta': False, 'pantalones': 'cortos', 'calzado': 'normal'}

if temp <= 20:
    dic['chaqueta'] = True
    dic['pantalones'] = 'largos'
    if prob > 75:
        dic['calzado']= 'invierno'
    print(dic)

elif temp > 20:
    if prob > 80:
        dic['calzado']= 'botas de agua'
    print(dic)
```

Fíjate que, en catalán, en la clave 'calzado', si escribimos '*botes d\'aigua*' lo hacemos con una barra delante. Esta es la sintaxis para que el intérprete identifique el apóstrofe como tal y no como una comilla que marca la finalización de la cadena de caracteres.

Las condicionales se pueden complicar o ser mucho más complejas, pero la estructura siempre será la misma. Recuerda siempre lo siguiente:

- La primera condición empieza con un *if*.
- Si no se cumple la primera y hay otra condición posible *elif* (puedes añadir tantos como necesites).

- Finalmente, escribirás *else* para ejecutar el código cuando no se cumpla ninguna de las condiciones previas.
- Las condiciones no van dentro de paréntesis.
- Las sentencias acaban en dos puntos.
- El código para ejecutar dentro del bloque de la condición tiene que ir indexando una posición a la derecha.

¡Ahora te toca a ti! No te rindas. Aún hay mucho más para aprender.

4.3 DOMINAR LOS CICLOS. SEGUNDO ALGORITMO

Los ciclos son estructuras que nos permite repetir los algoritmos tantas veces como se especifique en la condición marcada, o hasta que una condición determinada deje de ser válida. Principalmente, hay dos ciclos: los *for* y los *while*.

Los ciclos *for* tienen esta estructura básica:

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

En el inicio, siempre tenemos que utilizar la expresión *for* seguida de una nueva variable (en este caso, la denominaremos *i*, que irá alcanzando los valores que le marque una secuencia iterable). La secuencia iterable va precedida de la expresión *in*. Las acciones que estén dentro del bloque *for* (esas que están indexadas un espacio a la derecha después de dos puntos) se irán repitiendo hasta que *i* haya alcanzado todos los valores de la secuencia iterable.

En el ejemplo anterior, la función *range()* genera una lista de valores que van desde 0 hasta 9. La variable *i,* pues, coge el valor 0 y lo imprime; luego pasa lo mismo con el 1, el 2, el 3, y así sucesivamente hasta llegar al 9 (10-1).

Si la función *range()*, en lugar de tener dentro de su paréntesis el número 10 tuviera el 23, la secuencia generada iría del 0 al 22. El último valor de la secuencia que genera la función *range()* no está incluido. De hecho, la función *range()* puede tener un valor inicial (que sí se incluye en la secuencia) y que es 0 por defecto, sino se especifica su valor. Este sería un ejemplo:

```
for j in range(3,10):
    print(j)
```

En este ejemplo, la secuencia empieza en el 3 y termina en el 9. Fíjate que ahora la variable en cuestión se llama *j*. Puedes ponerle el nombre que quieras.

También podemos trabajar con secuencias iterables que ya tengamos en nuestro código, como por ejemplo las listas.

La estructura sería así:

```
numeros = [5,8,12,24]

for num in numeros:
    print(num)
```

La lista guarda en la variable *numeros* es iterada entera desde el primer número hasta el último. La nueva variable denominada *num* alcanzará todos los valores (de uno en uno) que hay en la lista y la función *print()* los imprimirá.

Fíjate, pues, que lo que estamos haciendo con el ciclo *for* es repetir la función *print()* hasta que hemos iterado todos los valores de la secuencia dada.

Dentro del bloque anterior, hemos utilizado solo la función *print()*, pero podemos añadir tantas líneas de código como queramos. Aquí exponemos algunas líneas más de código más complejas:

```
numeros = [5,8,12,24,13,4,6,11]
mayor10 = []
menor10 = []

for num in numeros:
    if num > 10:
        mayor10.append(num)
    else:
        menor10.append(num)

print('major10: ', mayor10)
print('menor10: ', menor10)
```

Este ejemplo nos muestra cómo podemos añadir, de la lista *numeros*, los valores mayores que 10 en una nueva lista denominada *mayor10* y, los valores menores que 10 en otra lista denominada *menor10*.

Los ciclos *while*, en lugar de repetir código siguiendo una secuencia iterable, lo que necesitan es que una condición sea verdadera para repetir el código. Cuando la condición deja de ser verdadera, el código deja de repetirse. Veamos un ejemplo muy sencillo:

```
n = 5
while n < 10:
    print(n)
    n = n + 1
```

En este caso, no hay ninguna secuencia iterable, sino que aparece una condición que dice: *mientras* n *sea menor que 10, ejecuta el código*. Este código imprime el valor actual de *n* y, a continuación, suma otra unidad a *n*. Cuando *n* alcanza el valor de 10, el código no se ejecutará más.

Nota: aunque, dentro del código, la condición deje de ser cierta, primero se ejecutará todo el bloque y, cuando sea el momento de empezar de nuevo el ciclo, el código no se ejecutará. Fíjate en el ejemplo siguiente. La suma de *n* + 1 va antes de la función *print()* pero el resultado es el mismo. En el último ciclo, cundo *n* es igual que 10, la condición deja de ser cierta antes que la función *print()*, pero el ciclo se detendrá cuando se haya completado todo el bloque. Eso quiere decir que la condición se pone a prueba solo al inicio (a la línea del *while*) y no durante el bloque.

```
n = 5
while n < 10:
    n = n + 1
    print(n)
```

Con los ciclos *while*, podemos añadir códigos tan complejos como haga falta. Fíjate en este ejemplo:

```
nombres = ['Joan', 'Maria', 'Enric', 'Silvia', 'Aleix']
j = 0
encontrado = False

while nombres[j] != 'Silvia':
    print('¿Hemos encontrado a Silvia?', encontrado)
    j = j + 1

encontrado = True
print('¿Hemos encontrado a Silvia?', encontrado)
```

Aquí iremos repitiendo el código hasta que encontremos el nombre *Silvia*.

Nota: el operador comparativo*!*= significa NO IGUAL. Recuerda que el operador comparativo == significa IGUAL.

4.4 REPASO DE LAS ESTRUCTURAS *FOR* Y *WHILE*

¡Hola!

Los ciclos son herramientas muy potentes que nos permiten resolver problemas con mucha agilidad. Son estructuras simples, sí, pero los bloques de código que contienen puede llegar a ser muy complejos. No hay límites. De hecho, en programación no hay límites. Puedes llegar a desarrollar un código muy complejo, incluso con los conocimientos que has ido adquiriendo hasta ahora con este curso.

Ahora repasaremos y trabajaremos con las estructuras *for* y *while*. Recuerda que las estructuras *for* recorren una secuencia iterable y las estructuras *while* se ejecutan mientras una condición sea cierta.

Si tenemos este ejemplo:

```
for i in range(2,20,2):
    print(i)
```

Aquí vemos que la función *range()* incorpora un tercer elemento. Este tercer número marca como se avanzará en la secuencia desde el 2 hasta el 19. Recuerda que el 20 no está incluido. Si ejecutamos el código, vemos que no se imprimen todos los números, sino que aparecen cada dos. Es justamente eso lo que indica el tercer número. Si, por ejemplo, el tercer elemento de la función *range()* es un 3 en lugar de un 2, entonces los números se imprimirán cada tres.

Y, ahora, empezaremos desde el 20 para acabar con el 3. ¿Cómo lo podemos hacer?

```
for i in range(20,2,-1):
    print(i)
```

Ahora, el primer elemento es el 20 (inicio), el segundo es el 2 (final y no incluido) y el tercero nos marca cómo avanzar por la secuencia. Como el tercer elemento es negativo, el sentido será a la inversa. Y, si en lugar de tener un -1 tenemos un -2, entonces los números aparecerán cada dos.

```
for i in range(20,2,-2):
    print(i)
```

Vamos a hacer algo divertido con el *while*.

```
import random

n = 0
ciclos = 0
while n != 5:
    print('n no es 5')
    n = random.randint(0,10)
    ciclos = ciclos + 1

print('¡Viva, hemos encontrado el 5 en el ciclo!', ciclos)
```

Aquí, lo que queremos ver es cuantos ciclos necesita el sistema para encontrar aleatoriamente el número 5. Hasta que no lo encuentre, repetirá el código y, cuando finalmente lo tenga, nos dirá cuántos ciclos han hecho falta.

Para poder generar un elemento aleatoriamente, debemos importar un módulo denominado *random* y hacer uso de una función que genera aleatoriamente un valor entre 0 y 10.

Como hemos dicho antes, las estructuras *for* y *while* son muy sencillas, pero el código que contienen en su bloque puede llegar a ser muy complejo.

¡Seguimos!

4.5 AHORA TE TOCA A TI: RETO. SER O NO SER

Genera una secuencia iterable formada por una lista de 10 números enteros mayor que 0 y menores que 20. Procura que todos los números de esta lista los entres por teclado. Una vez tengas la lista, realiza las acciones siguientes:

- Haz que todos los elementos sean multiplicados por 3, si son menores que 10, o por 2, si son mayores que 10. Haz uso de la estructura *for*.
- Imprime la lista. Haz uso de la estructura *for*.
- A continuación, imprime solo los cinco primeros números. Haz uso de la estructura *while*.

Si quieres, de forma opcional, intenta hacer este último paso:

Vuelve a realizar las acciones anteriores, pero en lugar de introducir los números por teclado, hazlo importando el módulo *random* y haciendo uso de la función *randint()*.

4.6 SOLUCIÓN EJERCICIO

¿Has conseguido resolver el ejercicio? ¡Felicidades! Si no ha sido así, no pasa nada. Si es la primera vez que programas, es muy normal que te resulte un poco complejo. La clave para llegar a programar es practicar mucho y reducir los problemas complejos a una colección de problemas más pequeños y sencillos. Aquí tienes el código (aunque seguramente habrá más opciones, como pasa siempre en programación). A continuación, lo explicamos paso a paso:

```
print('Introduce 10 numeros')
lista = []
for i in range(0,10):
    print('imprime posicion:', i)
    llista.append(int(input()))

print(lista)

for j in range(0, len(lista)):
    if lista[j] > 10:
        lista[j] = lista[j] * 2
    if lista[j] < 10:
        lista[j] = lista[j] * 3

print(lista)

j = 0
while j < 5:
    print(lista[j])
    j = j + 1
```

Primero, generamos una lista vacía y la vamos llenando con los valores que entramos por teclado. Fíjate que aquí tenemos el primer ciclo *for*. Como necesitamos 10 valores, lo que creamos es un ciclo con una secuencia iterable generada por la función *range()*. En cada ciclo, el sistema nos indica que debemos introducir un valor por teclado a la posición correspondiente. Recuerda que las entradas son *strings* y que necesitamos cambiarlas a números enteros. Por eso, utilizamos la función *int()*. Una vez tenemos la lista llena, decimos a la herramienta que la imprima. Bien, ya tenemos una parte del ejercicio resuelto.

La segunda parte nos dice que hemos de cambiar los valores de los elementos de la lista. Si son mayores que 10 los debemos multiplicar por 2 y, si son más pequeños, por 3. Aquí, la clave es saber que, para cambiar el valor de una lista, necesitamos acceder a ella, y eso se consigue mediante su posición. Por tanto, hemos de generar una secuencia iterable que corresponda a las posiciones de la lista. La función *range()*, pues, la escribiremos de tal forma que el primer valor sea 0 y el último sea 9. Puedes escribir este valor final a mano, poniendo explícitamente 10 (recuerda que la última posición, entonces, sería 10-1) o, como en el código que mostramos, haciendo uso de la función *len()*, que también nos devolverá el valor 10. A partir de aquí, cada valor de la secuencia es evaluado y multiplicado por 3 o por 2, dependiendo de si es mayor o menor que 10. Si el número fuera 10, no haría nada. Imprimimos la lista, para comprobar si nuestro código cambia sus valores, según los requisitos del problema.

Para el tercer paso, que nos dice que solo debemos imprimir los cinco primeros valores, necesitaremos que la condición que acompaña el *while* ponga fin al ciclo cuando *j* sea igual que 5. Mientras *j* (en este caso corresponde a la posición de la lista) sea inferior a 5, el ciclo volverá a empezar.

Nota: es muy importante que al final del bloque sumemos a *j* una unidad (j=j+1). Si no lo hacemos, el ciclo se repetirá eternamente, ya que la condición para poner fin al ciclo siempre será verdadera y, por tanto, nunca se detendrá.

Para resolver el ejercicio generando la lista de 10 números de manera aleatoria en lugar de introducir los valores a mano, el código cambiaría de esta manera. Fíjate que el cambio es realmente de solo dos líneas:

```
import random

print('Introduce 10 numeros')
lista = []
for i in range(0,10):
    lista.append(random.randint(1,19))

print(lista)

for j in range(0, len(lista)):
    if lista[j] > 10:
        lista[j] = lista[j] * 2
    if lista[j] < 10:
        lista[j] = lista[j] * 3
```

```
print(lista)

j = 0
while j < 5:
    print(lista[j])
    j = j + 1
```

Lo que hemos hecho ha sido añadir la primera línea de código para importar el módulo *random*. Después, solo hemos necesitado cambiar lo que hay dentro de los paréntesis de la función *append()* introduciendo la función *randint()*.

4.7 IDEAS CLAVE: ESTRUCTURAS CONDICIONALES Y CICLOS

En este módulo hemos aprendido a hacer uso de los ciclos *for* y *while*. Por un lado, los ciclos *for* nos permiten repetir bloques de código a través de una secuencia iterable. Por otro lado, podemos ejecutar el código que hay dentro de la estructura *while* repetidamente, siempre que la condición que marquemos sea cierta.

Los bloques de código, tanto para los ciclos *for* como para los ciclos *while*, son aquellas líneas de código que van después de los : - dos puntos – y que están indexadas una posición a la derecha. De hecho, cualquier bloque de código en Python que forme parte de una estructura siempre se escribirá así.

La función *range()* nos permite generar una secuencia iterable para ser utilizada en los ciclos *for*. El primer valor marca el inicio de la secuencia; el segundo marca el final (el valor no está incluido) y el tercero marca el avance a través de esta secuencia.

La sentencia *in* es la que nos indicia sobre qué secuencia se ejecutará el ciclo *for*. De hecho, la expresión *in* también puede referirse directamente a una lista y tratarla como una secuencia iterable.

También hemos aprendido a generar una secuencia de valores enteros de manera aleatoria haciendo uso de la función *randint()* del módulo *random*.

La función *while* evalúa la condición solo al inicio del ciclo. Eso quiere decir que, si durante el bloque de código no se cumple la condición, primero se ejecutará todo el bloque y se detendrá la repetición solo cuando el ciclo vuelva a empezar.

5 CONSTRUCCIÓN DE FUNCIONES

Las funciones permiten replicar bloques de código allí donde los necesitemos. Eso quiere decir que solo las tendremos que definir una vez para utilizarlas después sin límite de réplicas. Las funciones pueden contener, dentro de sus bloques, todas las sentencias y estructuras que hemos aprendido durante el curso.

Las funciones pueden ser simples o muy complejas e, incluso, se pueden utilizar una dentro de otra. Imagina que quieres crear un juego en el que el protagonista es un avatar que debe superar diversas aventuras. Todas las acciones que podrá hacer este avatar estarán definidas por las funciones que tú definas. Saltar, ir a la izquierda, atacar ... todo ello son funciones.

Las funciones se pueden almacenar en una variable. Sí, y también son objetos. Recuerda: en Python, todo son objetos.

5.1 APRENDRER A CONSTRUIR FUNCIONES Y CÓMO INVOCARLAS

Una función es un bloque de código reutilizable. Su misión principal es generar una acción o un cambio y que éste se pueda llevar a cabo sólo invocando la función sin tener la necesidad de volver escribir el código. Este es un ejemplo sencillo:

```
def duplicar(x):
    y = x * 2
    print (y)
```

Esta función se encarga de imprimir el doble de *x*. Analicemos la composición de una función:

- La sentencia *def* define que el objeto es una función.
- *duplicar* es el nombre de la función, aunque podría ser cualquier otro.
- (x) es un parámetro que lee la función y después lo utiliza en el código.
- *print()* es la acción que en este caso realiza la función.

Sí, una función puede estar dentro de otra función. En este caso, la función *print()* ya viene configurada con el paquete Python y no hay que volver a picar todo el código. Basta invocarla dentro de la nueva función, en este caso, la función *duplicar*.

Hay que decir, sin embargo, que la función, de momento, no genera ningún cambio ni ejecuta nada. Para que se ejecute el código de la función, es necesario invocarla. Para ello, sólo será necesario escribir:

```
duplicar(5)
```

Ahora, si ves el resultado en la terminal, el resultado que aparece es 10. Cuando una función se invoca, se asignarán a los parámetros sus valores, a fin de que se utilicen durante la ejecución del código.

Verás que el código de la función empaqueta al igual que los condicionales o los ciclos. Se ponen los dos puntos (:) en la primera línea y, a continuación, el código se indexa una posición a la derecha. El código que puede contener una función puede ser tan complejo como sea necesario.

Vamos a escribir algo más interesante.

Nota: todo aquello que ya no esté indexado dentro de la función, no formará parte de ella. El condicional *while* no es parte de la función *encadenar_palabras*.

```
mayores_cinco = []
menores_igual_cinco = []
replicas = 0


def encadenar_palabras(num):
    if num > 5:
        mayores_cinco.append(num)
    else:
        menores_igual_cinco.append(num)


while replicas < 5:
    print('introduce numero')
    encadenar_palabras(int(input()))
    replicas += 1


print('numero replicas es', str(replicas))
print('numeros mayores 5', mayores_cinco)
```

Este ejemplo muestra lo útil que es utilizar funciones. La función *encadenar_palabras()* añade el valor de su parámetro en la lista *mayores_cinc* o *menores_igual_cinco*, dependiendo de si este es mayor que 5 o no. Con el fin de no tener que repetir los condicionales y funciones *append()* durante el ciclo *while*, diseñamos primero la función y luego sólo hay invocarla dentro del bloque *while*. Esta función será invocada tantas veces como ciclos se hagan hasta que la condición de *while* deje de cumplirse. Observa que el valor del parámetro *num* alcanza el valor que introducimos por teclado a través de *input()*.

A continuación, redactaremos una función que sea capaz de devolver un valor. Esto significa que, cada vez que ejecutamos la función, ésta podrá emitir un resultado. La manera que tiene una función de generar un resultado es mediante la sentencia *return*. Veamos un ejemplo sencillo:

```
def duplicar(x):
    return x * 2

y = duplicar(3)

print(y)
```

Hemos utilizado la misma función que al inicio. Ahora, sin embargo, la función genera un resultado (*x* multiplicado por 2), que se puede asignar después a una variable. Aquí hemos escogido una variable *y*. Por lo tanto, cada vez que llamamos o invocamos la función, asignando a su parámetro *x* un valor, generaremos un resultado exportable a través de la sentencia *return*.

La invocación de una función con sentencia *return* se puede usar en cualquier otra operación y el valor o información que entrará en juego será, de hecho, aquel valor que genere esta operación. Por ejemplo, podemos modificar el código anterior de esta manera:

```
def duplicar(x):
    return x * 2

y = 5
z = y + duplicar(3)
print(z)
```

Ahora, la variable *y* tiene asignado el valor 5 (número entero). La variable *z* es la suma de la variable *y*, es decir 5, y la del valor que genera la función duplicar, es decir 6. El resultado de la operación, pues, es 5 + 6. Este resultado se guarda en la variable llamada *z*.

Las funciones pueden generar cualquier tipo de objeto. Por ejemplo, también podemos generar una lista. Este ejemplo es muy ilustrativo:

```
a = [4,2,7]
b = [6,3,12]

def lista_doble(lista):
    nueva_lista = []
    for i in range(len(lista)):
        nueva_lista.append(llista[i] * 2)

    return nueva_lista

print(lista_doble(a))
print(lista_doble(b))
```

Fíjate que el parámetro que toma la función es una lista. Podemos invocar la función, por ejemplo, con una lista que llamamos *a* y con otra que llamamos *b*. Podrás reutilizar la función tantas veces como quieras sin necesidad de cambiar el código.

5.2 CÓMO CONSTRUIR UNA FUNCIÓN Y CÓMO DEVOLVER UN BUEN RESULTADO

¡Hola de nuevo!

En la última parte del artículo anterior has aprendido a invocar funciones, asignando valor a su parámetro. Ahora construiremos una función con más de un parámetro, aunque no tiene mucha complicación, ya que se hace de la misma manera que si tuviéramos sólo uno. De hecho, una función puede coger tantos parámetros como se necesiten.

```
def duplicar_coordenadas(x,y,z):
    j = x
    x = y
    y = z
    z = j

    print('x:', x, 'y:', y, 'z:', j)

duplicar_coordenadas(4,7,9)
```

Aquí, básicamente, hemos hecho un intercambio entre tres parámetros. Es decir, cuando se invoca una función, los parámetros *x*, *y*, *z* tienen los valores 4, 7 y 9 respectivamente y, después, este son intercambiados según el código.

Ahora, vamos a hacer una cosa más divertida y que se acerca más a un desarrollo de código de un proyecto real. Ahora haremos que el retorno de una función sea el valor del parámetro de otra función:

```
def operacion_1(x):
    return x*3 + 6

print(operacion_1(6))

def operacion_2(x):
    print(x/3 - 6)

print(operacion_2(operacion_1(6)))
```

Quizás ves el código un poco complejo, si nunca has programado. Si es así, significa que estás ante algo nuevo y que sólo necesitas un poco de tiempo. Repasemos el código que hemos escrito.

La primera parte, la función *operacion_1*, es sencilla. Es una función que genera un valor que es el triple del valor que alcanza el parámetro *x* más 6. Si invocamos la función *operacion_1* dentro de la función *print()* y a *x* le damos el valor de 6, el resultado que obtendremos será 24.

Después, definimos la función *operacion_2*, que tampoco tiene mucha complicación. Lo que hace es, de un valor *x*, volver otro valor que es el resultado de dividir por 3 y restarle 6. Ahora, es aquí cuando llega la parte más crítica.

Cuando invocamos la función *operacion_2* dentro de la función *print(),* el valor del parámetro será 24, que es el resultado de la función *operacion_1*. La invocación de la función alcanzará el valor que se genere con la sentencia *return* y, por tanto, este valor podrá ser asignado al parámetro de la función *operacion_2*.

Recuerda que, en este caso, la función *print()* imprime un número con decimales, porque es el resultado de una división. Si, por alguna razón, quisiéramos que fuera un número entero, deberíamos sustituir la barra diagonal por una de doble. El código sería así:

```
print(x//3 - 6)
```

Antes de terminar este vídeo, podríamos hacer una última cosa. Recuerda que en Python todo son objetos y, por tanto, las funciones también. Esto significa que podemos guardar la función entera en una variable (y no sólo el resultado que genera).

```
def operacion_1(x):
    return x*3 + 6

f = operacion_1

print(f(4))
```

Ahora la función *operacion_1* queda guardada en una variable *f* y podemos invocar a través de su variable y asignar el valor del parámetro de la misma manera.

¡Ahora sí que has dado un paso adelante!

Continuamos.

5.3 AHORA TE TOCA A TI: RETO. HAZLO UNA VEZ Y LLÁMALA TANTAS VECES COMO QUIERAS

A continuación, te proponemos resolver un ejercicio vinculado a las funciones.

Supongamos que tenemos una colección de números enteros guardados en una lista. Construye una función que tome como parámetro esta lista y que vuelva una nueva lista los valores de la que sean el doble de la lista de entrada.

Por ejemplo, si el parámetro de la función es [4,6,8], el retorno será [8,12,16].

Construye una nueva función (que nombraremos función_1) que coja como parámetro y que, primero, multiplique por 3 sus valores, si estos no son superiores a 10 y, después, que imprima la lista. Invoca esta función de tal manera que el valor de su parámetro sea la lista que vuelva laprimera función.

Resuelve el ejercicio por estas tres listas diferentes que cogen la *funcion_1* como parámetro:

[4,5,7]

[1,3,2]

[3,6,8]

Procura resolver el ejercicio suponiendo que la lista que toma como parámetro la *funcion_1* es una secuencia de cuatro valores enteros generados aleatoriamente del 1 al 9.

5.5 SOLUCIÓN EJERCICIO

Antes de solucionar el ejercicio, es importante leer el enunciado y analizado qué herramientas de código nos serán útiles. En este caso, por ejemplo, y una vez hayamos visto que tenemos que transformar una lista, lo más habitual es pensar que los ciclos *for* nos serán de gran ayuda. Aquí se muestra el código que soluciona el ejercicio para las tres listas dadas [4,5,7], [1,3,2] y [3,6,8]:

```
seq = [4,5,7]

def funcioN_1(lista):
    for i in range(len(lista)):
        lista[i] = lista[i] * 2

    return lista


def funcion_2(lista):
    for i in range(len(lista)):
        if lista[i] <= 10:
            lista[i] = lista[i] * 3
    print(lista)


funcion_2(funcion_1(seq))
```

Siempre que necesitamos recorrer una secuencia y cambiar sus valores, tendremos que acceder a todas sus posiciones. Y, una vez estemos en la posición correcta, podremos hacer el cambio de valor.

La *funcion_1* recorre toda la lista y multiplica por 2 todos sus valores. Y, después, usamos la sentencia *return* para que la función genere como resultado la nueva lista, cuando sea invocada.

La *funcion_2* hace exactamente lo mismo que la *funcion_1* pero, en lugar de multiplicar por 2, lo hace por 3, aunque antes impone una condición: en aquellos números de la lista de entrada que sean superiores a 10 no se llevará a cabo su multiplicación por 3.

Como el ejercicio nos obliga a que el valor del parámetro de la *funcion_2* sea el valor que genera la *funcion_1*, tendremos que invocar la *funcion_1* dentro de los paréntesis de la *funcion_2*. También podríamos guardar previamente la lista que genera la *funcion_1* en una variable y pasarla como valor al parámetro.

Con el fin de resolver la segunda parte del ejercicio, la que nos pide que el parámetro de la *funcion_1* sea una secuencia de cuatro valores enteros de 1 a 9 generados aleatoriamente, necesitamos este código:

```
import random
```

```
def funcion_1(lista):
    for i in range(len(lista)):
        lista[i] = lista[i] * 2

    return lista


def funcion_2(lista):
    for i in range(len(lista)):
        if lista[i] <= 10:
            lista[i] = lista[i] * 3
    print(lista)


def lista_aleatoria(n):
    lista = []
    for i in range(n):
        lista.append(random.randint(1,9))
    print('lista aleatoria: ', lista)
    return lista


funcion_2(funcion_1(lista_aletoria(4)))
```

La primera diferencia que vemos es que necesitamos importar el módulo *random* para poder utilizar la función *randint()*, que nos permitirá generar los valores aleatorios. Y, a fin de generar una lista con estos valores aleatorios, necesitamos escribir una función que genere esta lista y que coja un parámetro que nos indique el número de valores (en este caso, el ejercicio nos pide cuatro valores). En el código de arriba, esta función es *lista_aleatoria*. Observa que, para generar la lista, necesitamos, primero, declarar una que esté vacía y, después, ir añadiendo los valores hasta tenerlos todos cuatro.

La última línea del código se encarga de invocar la *funcion_2*, el parámetro es la lista que genera la *funcion_1*. Fíjate, pues, que el parámetro de la *funcion_1* es la lista de valores aleatorios que ha generado la función *lista_aleatoria*.

Durante la redacción de código profesional, es muy normal encontrar estas invocaciones unas encima de las otras. Es cuestión de práctica acostumbrarse a trabajar así.

5.6 IDEAS CLAVE: CONSTRUCCIÓN DE FUNCIONES

Ya hemos terminado el curso. Repasamos las ideas más importantes que han aparecido relacionadas con las funciones.

Con el fin de construir funciones, primero necesitamos escribir el bloque y luego invocarlas para usarlas.

La construcción se compone de la sentencia *def*, el nombre de la función, los parámetros y el código. El código que forma parte de esta función siempre debe ir después de dos puntos (:) y estar indexado una posición a la derecha.

Las funciones pueden generar valores mediante la sentencia *return*. Esto significa que una función, cuando sea invocada, enviará un dato que será lo que vuelva la sentencia *return*. Este dato se puede guardar en una variable, para usarla después, directamente dentro de otra operación o como parámetro de otra función. De hecho, las funciones pueden ser invocadas unas sobre las otras, haciéndose pasar por parámetros.

Las funciones enteras pueden ser guardadas en una variable. Esta se puede invocar con los parámetros que correspondan, a fin de llevar a cabo las acciones de la función.

Descubre todo lo que Barcelona Activa puede hacer por ti

Acompañamiento durante todo el proceso de búsqueda de empleo

barcelonactiva.cat/treball

Apoyo en la puesta en marcha de tu idea de negocio

barcelonactiva.cat/emprenedoria

Servicios a las empresas e iniciativas socioempresariales

barcelonactiva.cat/empreses

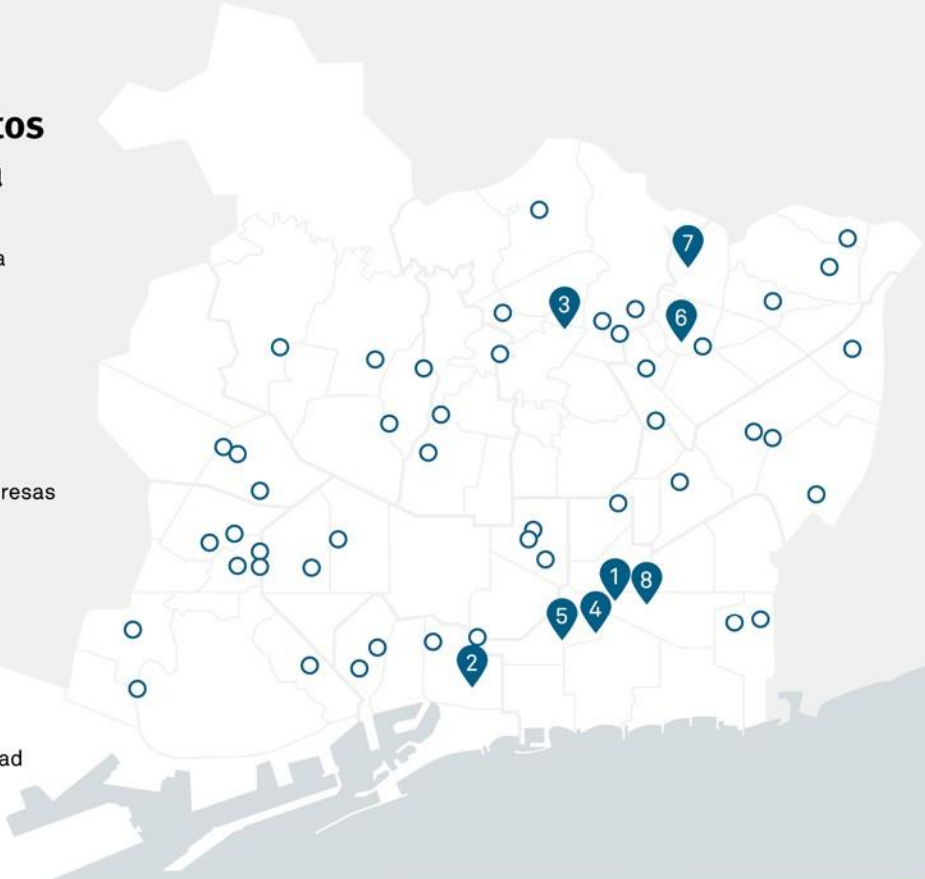
Formación tecnológica y gratuita para la ciudadanía

barcelonactiva.cat/cibernarium

Red de equipamientos de Barcelona Activa

1. Sede Central Barcelona Activa
   Porta 22
   Centro para la Iniciativa Emprendedora Glòries
   Incubadora Glòries
2. Convent de Sant Agustí
3. Ca n'Andalet
4. Oficina de Atención a las Empresas
   Cibernàrium
   Incubadora MediaTIC
5. Incubadora Almogàvers
6. Parque Tecnológico
7. Nou Barris Activa
8. innoBA

○ Puntos de atención en la ciudad



Barcelona Activa

Ajuntament de Barcelona

Última actualización 2022

Cofinanciado por:

UNIÓ EUROPEA
Fons Europeu de Desenvolupament Regional

Síguenos en las redes sociales:

barcelonactiva.cat/cibernarium
barcelonactiva
barcelonactiva
company/barcelona-activa