

PROGRAMACIÓN CON CÓDIGO: PYTHON

ELEMENTOS DE CONTROL (CONDICIONALES Y BUCLES)

Este documento es de uso único e intransferible para el alumno matriculado en el curso. Cualquier reproducción física o digital del documento sin permiso de los autores vulnera los derechos de propiedad intelectual de los mismos.

INDICE

INDICE.....	3
1. ALGORITMO.....	4
2. TRY - EXCEPT.....	7
3. ¿UN IGUAL O DOS IGUALES?.....	15
4. PUNTEROS.....	16
4.1 True y False.....	23
5. CONDICIONALES.....	24
5.1 ¿Sólo puedo comprobar igualdad?.....	30
6. OPERADORES LÓGICOS (AND Y OR).....	31
7. BUCLE WHILE.....	34
8. INTRODUCCIÓN A LAS LIBRERÍAS.....	39
9. BUCLE FOR.....	45
10. INSTRUCCIONES DE CÓDIGO USADAS.....	53
11. RETO.....	55

1. ALGORITMO

En el módulo anterior hemos realizado nuestros primeros algoritmos. Llamamos algoritmo a la sucesión de instrucciones de un programa que se ejecutan en un orden concreto. Un algoritmo humano podría ser una receta de cocina, o la secuencia de pasos para llegar a una dirección que entrega google maps al realizar una búsqueda:



Google maps tiene desarrollado un muy potente algoritmo, es capaz de mostrar a tiempo real el camino más eficiente para llegar a un destino, incluyendo la gestión del tráfico, por lo que la ruta es cambiante en función de la hora del día, condiciones meteorológicas, etc.

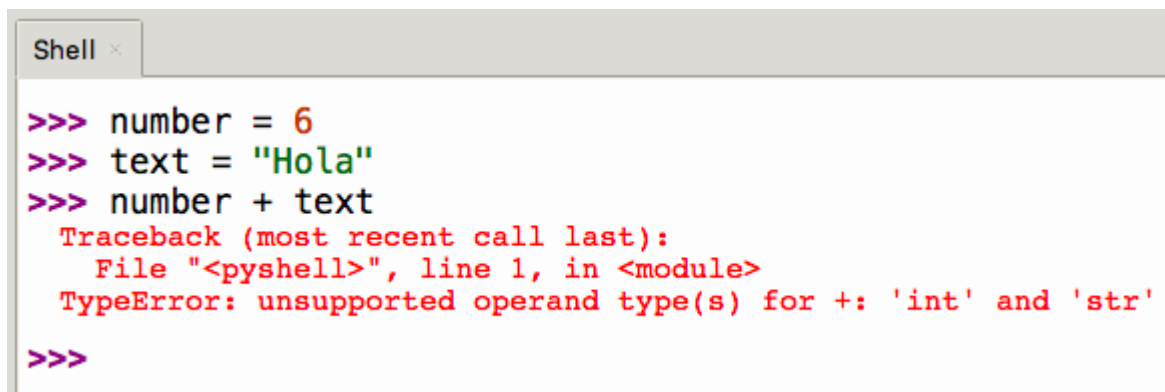
Obviamente, existen muchas formas de llegar a un destino. Una de ellas será la más eficiente, aunque puede que haya un par de soluciones de eficiencia similar. Luego habrá otras tantas buenas soluciones y, por último, habrá un sinfín de soluciones malas, malísimas o totalmente absurdas (como ir de Berlín a Praga pasando por Sidney).

La elaboración de un algoritmo es todo un desafío, pues siempre hay una multitud de posibilidades que pueden conseguir el efecto deseado pero no todas serán óptimas o adecuadas. Una persona capaz de hacer un buen algoritmo en Python es actualmente una persona con muchas posibilidades en el mercado laboral. Las empresas cotizan y se valoran casi más por sus algoritmos que por cualquier otra cosa. El algoritmo de Google, el de Facebook, el de Amazon... todos ellos son algoritmos muy bien desarrollados y con una soberbia forma de conseguir el efecto deseado.

Así mismo, un mal algoritmo puede, a priori, cumplir de forma correcta una función, pero a largo plazo traerá problemas y será difícil su adaptación a nuevos contextos, su modificación, su utilización más allá del caso concreto...

Una buena práctica es desarrollar la costumbre de pensar en el algoritmo en general y esquematizarlo antes de ponernos a programar. Así mismo, todo algoritmo tiene siempre lagunas o fallos que hay que corregir o mejorar. Al proceso según el cual se incorporan nuevas soluciones o se implementan mejoras en un algoritmo existente se le denomina **iteración**. Una iteración es un proceso de mejora de un algoritmo (en realidad no es algo que se aplique exclusivamente a los algoritmos) que engloba todo el proceso que va desde una versión ya funcional a otra versión terminada que mejora la anterior en alguna o algunas funcionalidades.

En los algoritmos creados en la unidad 2 había posibilidad de cometer errores como, por ejemplo, tratar de sumar una variable que contenía un entero (*int*) con una variable que contenía un string (*str*). No sé si lo probaste, pero puedes probarlo y verás algo parecido a esto:

A screenshot of a Python Shell window. The window has a title bar with a close button and the text 'Shell'. Inside the window, the following code is entered:

```
>>> number = 6
>>> text = "Hola"
>>> number + text
```

The last line of code has triggered an error. The error message is displayed in red text:

```
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

The prompt '>>>' is shown again at the bottom of the window.

Un buen algoritmo debe estar desarrollado de tal forma que **jamás** pueda incurrir en un error, los errores suponen la caída del intérprete de Python, lo cual obliga a reiniciar el programa.

Una caída del intérprete puede no parecer muy grave en la tranquilidad de nuestro aprendizaje de los niveles más básicos de Python, pero es una catástrofe mayúscula en un algoritmo profesional como podría ser el de Google Maps. Imagina que introduces a buscar en Google Maps un lugar que produce una caída del intérprete y, de golpe, se cae todo el sistema de navegación de Google en una buena parte de tu área geográfica... En Palo Alto rodarían cabezas... (Palo Alto es una zona de California donde muchas de las empresas tecnológicas punteras de Estados Unidos ubican su sede principal).

Por ello vamos a empezar a trabajar en mejorar nuestros algoritmos para que no se puedan producir errores, o al menos de momento minimizar la posibilidad de que ocurran.

2. TRY - EXCEPT

Nuestros primeros programas con Python han constado de un algoritmo secuencial que no alteraba su orden de ejecución, se ejecutaban paso a paso y siempre de la misma forma.

En este tema, vamos a ver **elementos de control** para nuestro script, mediante los cuales el algoritmo seguirá un camino u otro en función de comprobaciones y decisiones que realizará Python, siempre respondiendo a nuestra programación.

Antes de lanzarnos a ver cómo se usan esos elementos de control vamos a aprender una instrucción de código bastante interesante. En el tema anterior hemos visto ya que en ocasiones Python no sabe cómo realizar una instrucción y nos arroja un error. En muchas ocasiones el programa puede estar bien realizado pero el usuario no ha sabido introducir un input adecuado. Por ejemplo, si pedimos un número y contesta una letra el programa no va a saber tratar esa letra como número.

Vamos a realizar un pequeño script donde pidamos al usuario dos números y los vamos a sumar. Guarda en un documento .py el siguiente código y ejecútalo por terminal o con la IDLE. Si hay algo que no entiendas del mismo repasa el tema anterior:

```
print("Vamos a sumar dos números")
numero_1 = input("¿Cuál es el primer número?: ")
numero_2 = input("¿Cuál es el segundo número?: ")
numero_1 = int(numero_1)
numero_2 = int(numero_2)
suma = numero_1 + numero_2
suma = str(suma)
print("La suma de ambos números es " + suma)
```

Mucho cuidado si copiamos y pegamos texto para hacer un script, a veces ciertos caracteres se pegan de forma errónea en el archivo de destino, cosa que suele ocurrir con las comillas. Los editores de texto las escriben como comillas iniciales y finales y Python sólo entiende de comillas rectas (es así de especialito).

Al ejecutar el script anterior obtendremos algo como:

```
MacBook-Air-de-Alfredo-2:Desktop alfredosanchezsanchez$ python3 suma.py
Vamos a sumar dos números
¿Cuál es el primer número?: 5
¿Cuál es el segundo número?: 8
La suma de ambos números es 13
MacBook-Air-de-Alfredo-2:Desktop alfredosanchezsanchez$
```

¿Qué ocurre si elegimos un carácter no numérico?:

```
MacBook-Air-de-Alfredo-2:Desktop alfredosanchezsanchez$ python3 suma.py
Vamos a sumar dos números
¿Cuál es el primer número?: 3
¿Cuál es el segundo número?: a
Traceback (most recent call last):
  File "suma.py", line 5, in <module>
    numero_2 = int(numero_2)
ValueError: invalid literal for int() with base 10: 'a'
MacBook-Air-de-Alfredo-2:Desktop alfredosanchezsanchez$
```

Como puedes observar Python nos ha dado un error, no puede convertir una **a** en un **int**. Vamos a intentar evitar esos errores.

Python tiene una función que nos permite intentar realizar una instrucción y, en caso de que la ejecución de esa instrucción produzca un error ejecutará otra instrucción que señalemos para tal fin. Dicha instrucción se denomina *try - except*. Le pediremos a Python que pruebe a ejecutar un código (*try*) y si el resultado del código es un error pasará a ejecutar el código contenido en el *except*.

Llegados a este punto conviene hablar de las indentaciones como base de jerarquía en Python. Si no has entendido esta última frase no te preocupes, a lo largo del curso vas a tener claro que son las jerarquías.

Hasta ahora nuestros programas eran una secuencia de pasos ordenados, cada paso era único y no conllevaba varias acciones. Un paso era preguntar al usuario algo, otro paso era crear una variable, otro podría ser imprimir un resultado... Pero poco a poco vamos a ir viendo que una instrucción de código puede tener varias acciones alojadas dentro de ella, por ejemplo intenta convertir esta variable a una de tipo entero y si lo consigues súmale 7 y después imprímela. Es verdad que cada acción es una en sí misma, pero todas ellas se ejecutarán si es posible convertir la variable en una de tipo entero. Decimos entonces que todas esas instrucciones están **anidadas** en una instrucción.

En algunos lenguajes de programación, para marcar dónde empieza y dónde termina una función con cosas anidadas, se usan llaves ({ }). De esta forma el programa sabe perfectamente dónde empieza una instrucción porque se abre una llave y dónde termina la misma porque lo marca una llave final. Algo del estilo de:

```
si es martes {  
  revisión de obra en calle...  
  reunión de dirección facultativa  
}  
si es miércoles {  
  reunión con el grupo inversor  
  revisión de obra en calle...  
}
```

Python es bastante más simple y ello nos puede llevar a cometer errores si no sabemos cómo entenderle (o cómo nos entiende él a nosotros). Cuando una instrucción de código (como puede ser un *try*) contiene un código a ejecutar hay que indentar dicho código respecto de la instrucción. Algo así:

```
si es martes:
    revisión de obra en calle...
    reunión de dirección facultativa
si es miércoles:
    reunión con el grupo inversor
    revisión de obra en calle...
```

Con indentar nos referimos a situar toda instrucción anidada en una vertical más hacia la derecha que la instrucción a la que pertenece. Este sería el ejemplo de un código para que veas a lo que me refiero:

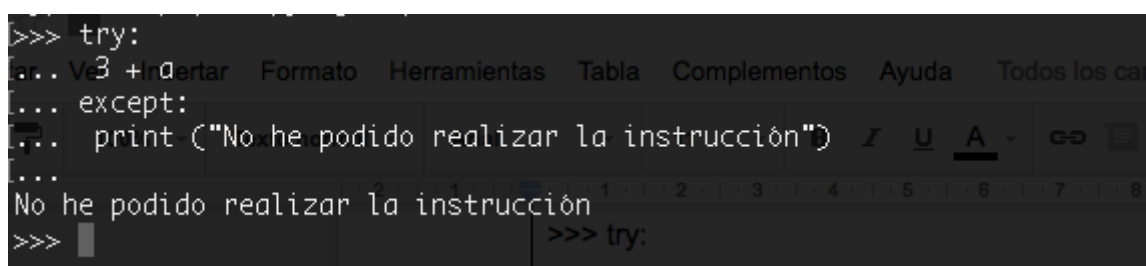
```
try:
    3 + a
except:
    print("No he podido realizar la instrucción")
```

Si te fijas, el código anidado en el *try* está indentado, es decir, desplazado a la derecha respecto de la posición vertical del *try*. Así mismo, el contenido del *except* está desplazado a la derecha. Todas las funciones de Python necesitan que su contenido esté indentado para que sepan qué contienen exactamente y dónde acaba su campo de acción.

La indentación, además, tiene que ser la misma para todas las instrucciones anidadas. No puede ser una un poco más a la derecha que otra, todas en la misma vertical si pertenecen a la misma anidación. Hay gente que indenta con un espacio a la derecha, otras personas lo hacen con varios espacios, otras con un tabulador...

Algunos programadores son bastante tiquismiquis con esto y creen que es imprescindible usar el tabulador, de manera que indentan tabulando el texto. De hecho, hay una serie muy divertida llamada Silicon Valley en la cual el protagonista termina una relación sólo porque su novia indentaba usando tres espacios en vez de una tabulación. La serie trata de reflejar fielmente todos los clichés de las startups tecnológicas y sus creadores, generalmente programadores.

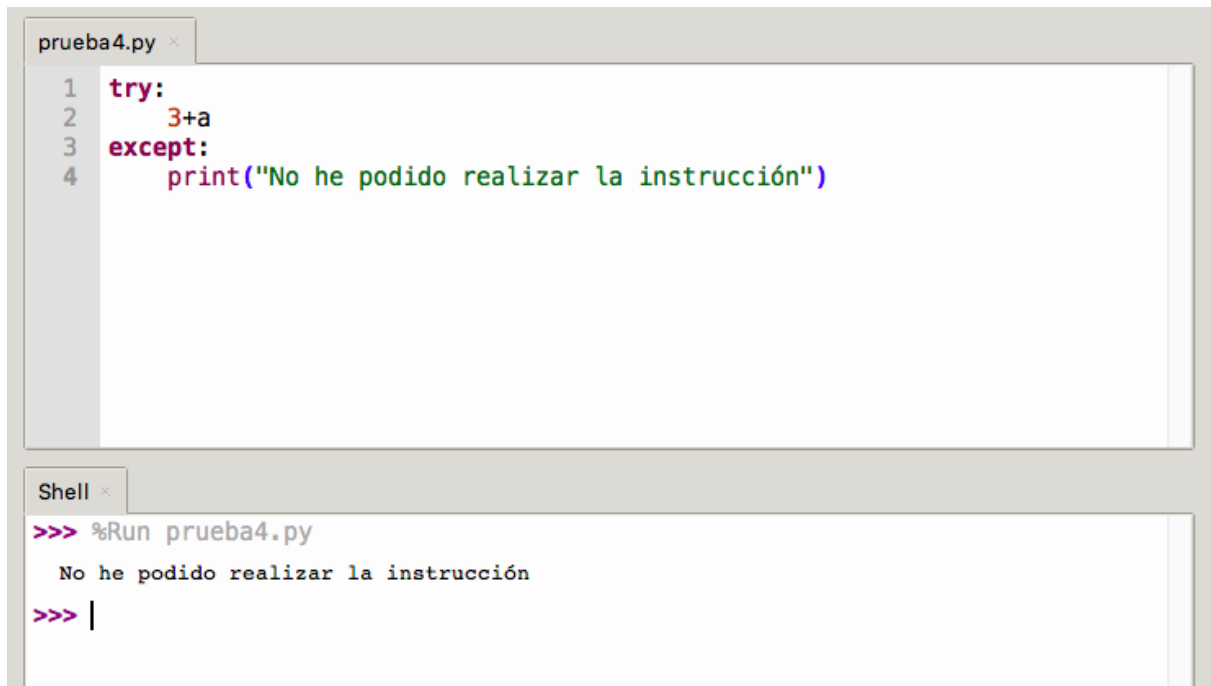
Recordemos la instrucción anterior y veamos cómo sería su ejecución a través de terminal:



```
>>> try:
... except:
... print('No he podido realizar la instrucción')
No he podido realizar la instrucción
>>>
```

La cantidad de espacios para indentar el contenido de una instrucción compuesta de Python es indiferente, puedo poner un espacio, tres, cuatro... lo importante es que siempre ponga la misma cantidad de espacios. En scripts lo habitual es usar una tabulación. En terminal o por la IDLE es un poco complicado este proceso las primeras veces porque marca con puntos suspensivos (como se puede ver en la imagen superior) que nos encontramos dentro de una instrucción y debemos poner espacios, la tecla de tabulación no funciona. En el ejemplo he puesto un único espacio bajo el *try* y un único espacio bajo el *except*.

En el intérprete (Shell) de Thonny no se puede escribir esta instrucción directamente. Puedes ejecutar la instrucción anterior en Thonny guardando el código y ejecutándolo como nuestro a continuación:



```
prueba4.py x
1 try:
2     3+a
3 except:
4     print("No he podido realizar la instrucción")

Shell x
>>> %Run prueba4.py
    No he podido realizar la instrucción
>>> |
```

Prueba a repetir el código de la imagen anterior y, tras ello, repítelo cambiando la `a` por un `5` para que veas que el código que contiene el `try` sí se ejecuta cuando derivado de intentar ejecutarlo no se produce un error.

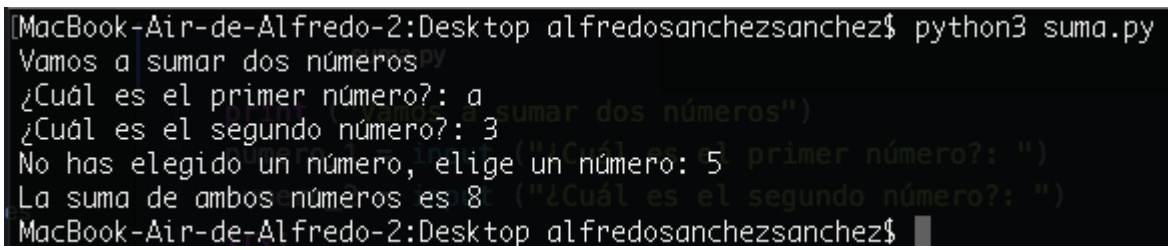
Una vez que hemos visto cómo funciona un `try - except` vamos a ver cómo podemos utilizar esta instrucción en nuestro script suma. Copia, prueba y analiza el siguiente código. Intenta entender todo lo que está ocurriendo antes de seguir leyendo.

```

print("Vamos a sumar dos números")
numero_1 = input("¿Cuál es el primer número?: ")
numero_2 = input("¿Cuál es el segundo número?: ")
try:
    numero_1 = int(numero_1)
except:
    numero_1 = input("No has elegido un número, elige un número: ")
    numero_1 = int(numero_1)
numero_2 = int(numero_2)
suma = numero_1 + numero_2
suma = str(suma)
print("La suma de ambos números es " + suma)

```

El programa está corregido para tratar de convertir el contenido de la variable *numero_1* en entero y, si no puede hacerlo, nos volverá a preguntar por un número y lo convertirá en número entero. Vamos a probarlo:



```

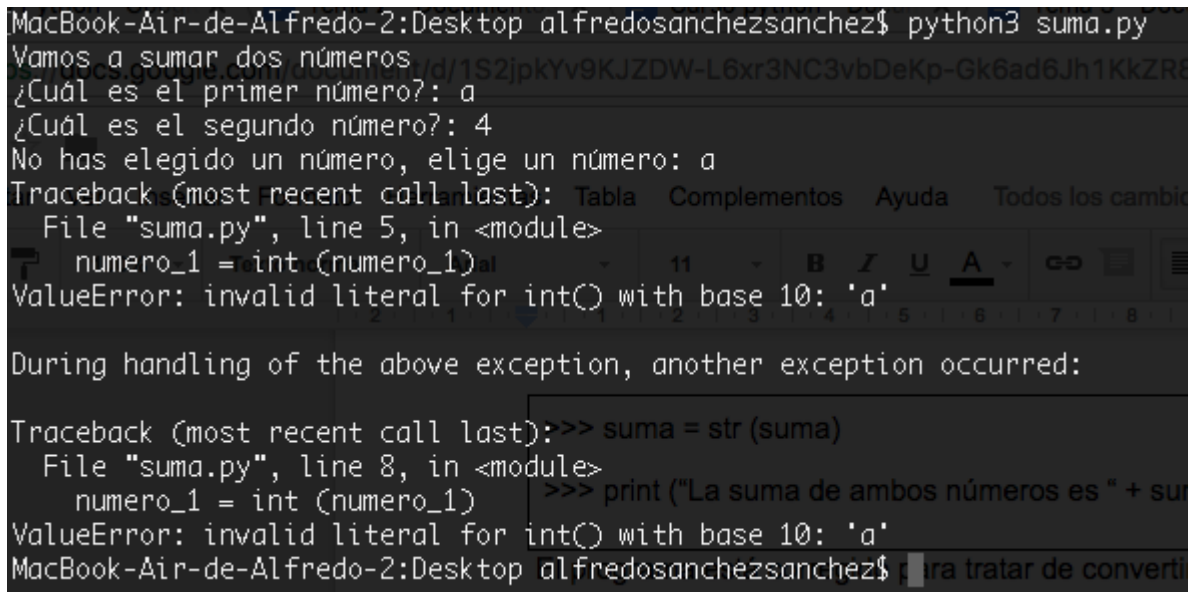
MacBook-Air-de-Alfredo-2:Desktop alfredosanchezsanchez$ python3 suma.py
Vamos a sumar dos números
¿Cuál es el primer número?: a
¿Cuál es el segundo número?: 3
No has elegido un número, elige un número: 5
La suma de ambos números es 8
MacBook-Air-de-Alfredo-2:Desktop alfredosanchezsanchez$

```

Si observas la ejecución del código verás que inicialmente seleccioné una **a** y un **3** y el programa me preguntó después por un número ya que ejecutar el *try* le suponía un error. Al escribir un 5 ya sí podía hacer la suma.

Antes de seguir, sitúa un *try - except* para la variable *numero_2* como hemos hecho para la variable *numero_1* y prueba el programa.

Si pensamos un poco en el recurso que hemos aprendido podríamos llegar a la conclusión que nos evita un error una única vez, pero en nuestro ejemplo el usuario podría seguir poniendo letras y volvería a ocurrir un error, como se observa en la siguiente imagen fruto de una ejecución:



```
MacBook-Air-de-Alfredo-2:Desktop alfredosanchezsanchez$ python3 suma.py
Vamos a sumar dos números
¿Cuál es el primer número?: a
¿Cuál es el segundo número?: 4
No has elegido un número, elige un número: a
Traceback (most recent call last):
  File "suma.py", line 5, in <module>
    numero_1 = int(numero_1)
ValueError: invalid literal for int() with base 10: 'a'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "suma.py", line 8, in <module>
    numero_1 = int(numero_1)
ValueError: invalid literal for int() with base 10: 'a'
MacBook-Air-de-Alfredo-2:Desktop alfredosanchezsanchez$
```

Como ves, al volver a elegir una **a** tras elegirla en primera opción ocurre un error y encima Python avisa que el error ocurrió durante la excepción, es decir, durante la ejecución de un *try - except*.

Más adelante aprenderemos a evitar un error reiterativo, pero de momento nos podemos conformar con dar una segunda oportunidad de introducir un número al usuario. Ojalá ésta fuese una solución válida, pero por desgracia el usuario tiende a realizar cualquier acción que pueda llevar a un error, y es algo que no ocurre conscientemente. En desarrollo de videojuegos existe un puesto laboral que se dedica a jugar el juego y hacer absolutamente todo lo posible en busca de bugs (bichos en inglés, término usado para identificar un error en un programa).

3. ¿UN IGUAL O DOS IGUALES?

Es importante tener claro a partir de ahora la diferencia entre usar un igual (=) y usar dos iguales (==).

Usar un igual (=) significa asignar un valor. La expresión `numero = 3` asigna a la variable `numero` el valor 3. Una expresión más lógica para este caso podría ser `numero ← 3`, es decir, introduce en `numero` el valor 3.

En cambio, usar dos iguales (==) significa comparar si dos elementos tienen el mismo valor. La expresión `numero == 3` comprueba si el contenido de la variable `numero` tiene el valor 3, es decir, busca si lo que se encuentra a la izquierda de la doble igualdad y lo que se encuentra a la derecha coinciden, valen lo mismo (ojo, esto no quiere decir que sean lo mismo, simplemente valen lo mismo).

Una comprobación con == en Python nos dará el valor `True` de ser cierta y el valor `False` de no ser cierta, puedes observar la siguiente captura de un código en terminal para entenderlo:

```
>>> numero = 3 numero = int (numero)
>>> numero == 3
True
>>> numero = 4 numero = input ("No has elegido un número, elige un n
>>> numero == 3 numero = int (numero)
False
>>> if numero%2 == 0:
>>>     print ("Has elegido un número par")
```

En el primer caso asigno el valor 3 a la variable `numero` y luego compruebo si la variable `numero` coincide con el número 3 y, al ser cierto, el programa nos devuelve el valor `True`. En el segundo caso asigno a la variable `numero` el valor 4 y luego compruebo si `numero` tiene el valor 3, con un resultado de `False` dado que no es cierta esa coincidencia, `numero` contiene el valor 4, no 3.

4. PUNTEROS

Python es un lenguaje muy interesante y tiene diferencias sustanciales con otros lenguajes.

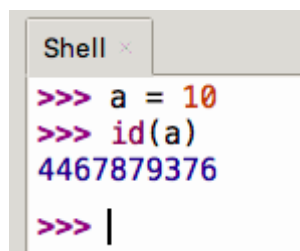
Ya hemos visto que al definir una variable estamos reservando un espacio en memoria para almacenar un dato y que ese espacio es variable en función del tipo de dato que albergará.

En otros lenguajes al definir una variable creamos algo conocido como puntero, que no es más que algo parecido a la dirección en memoria que tiene el contenido de esa variable. Esa dirección es fija para esa variable y se puede jugar con esos punteros en una programación más avanzada (de hecho se torna casi imprescindible conocer la forma de trabajar con punteros, es decir, direcciones en memoria de datos).

Un ejemplo de lo anterior sería C++, por decir un lenguaje de mayor bajo nivel que Python. Con bajo nivel nos referimos a que está más cerca del lenguaje máquina. El lenguaje máquina es aquel que entiende la máquina (básicamente, unos y ceros), cuanto más nos acerquemos a este lenguaje, diremos que dé más bajo nivel es un lenguaje.

Python, en cambio, es un lenguaje de alto nivel, está más lejos del lenguaje máquina y esto es así para conseguir que sea más fácil escribir el código y no haya que preocuparse de muchos pormenores.

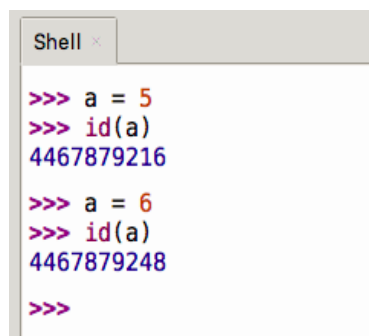
En Python también existen los llamados punteros, que apuntan a objetos en memoria, pero hay diferencias sustanciales con otros lenguajes. Quizá con los contenidos vistos hasta ahora todo esto sea un poco desconcertante y no sepas para qué narices sirve, pero es bueno conocerlo para posteriormente encontrarle sentido. Para conocer el puntero de una variable en Python debemos usar la función *id* (de identidad). Mira el siguiente código:



```
Shell x
>>> a = 10
>>> id(a)
4467879376
>>> |
```


Como ves, he creado una variable de nombre *a* y he almacenado en ella el dato *10*. Ya sabemos con esta información que *a* es una variable de tipo entero. Posteriormente he pedido que me muestre el puntero de *a* y vemos que dicho puntero es el número 4467879376. No vamos a entrar a analizar esa información, pero sí que sabemos que esa es una dirección de memoria.

Por la forma de funcionar de Python podemos decir que la palabra variable tendría poco sentido, pues es algo que varía, pero es interesante ver que Python, al cambiarle el valor a una variable no cambia el contenido de la misma, sino que crea una nueva con el nuevo contenido y el mismo nombre, por lo que cambia la identidad de la variable.



```
Shell x
>>> a = 5
>>> id(a)
4467879216

>>> a = 6
>>> id(a)
4467879248

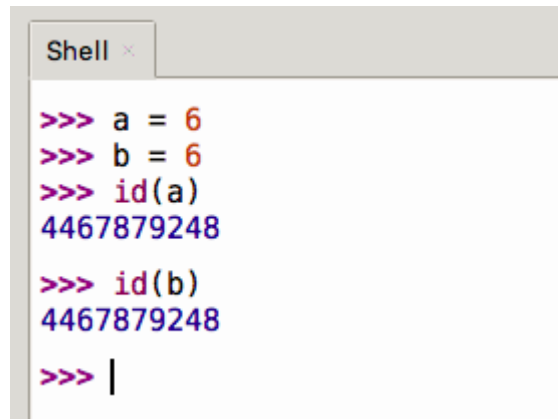
>>>
```

Fijaos que el puntero para *a* cuando contenía un 5 no es el mismo que cuando ha contenido un 6.

En lecciones sucesivas seguiremos profundizando en esto, pero podemos decir de momento que en Python hay objetos **mutables** y **no mutables**. Los objetos mutables son aquellos que pueden mutar, es decir, cambiar, y los no mutables no pueden cambiar. Las variables en Python son **no mutables**. No podemos cambiar el valor de una variable, podemos crear una nueva que se llame igual pero con otro valor. El efecto es el mismo que si cambiáramos el contenido, pero sustancialmente Python está haciendo algo bastante diferente.

Es curioso que algo que se llama variable no pueda cambiar... ¿verdad? Pero al final hace la programación más sencilla porque no te tienes que preocupar de lugares de memoria y espacios en memoria, sólo programar.

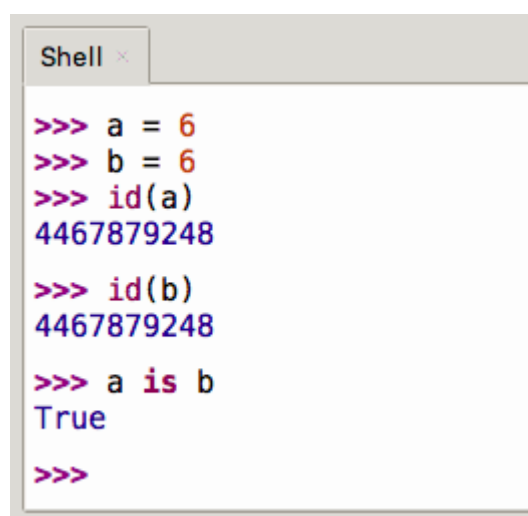
Eso sí, Python también hace cosas eficientes, por ejemplo, asignar identidades iguales a variables con diferente nombre pero mismo contenido:



```
Shell x
>>> a = 6
>>> b = 6
>>> id(a)
4467879248
>>> id(b)
4467879248
>>> |
```

¿Y por qué cuento todo esto? Pues porque antes he dicho que valer lo mismo no es lo mismo que ser lo mismo, pero en el caso de las variables el que valgan lo mismo, en Python, si significa que son lo mismo.

Para saber si dos objetos son lo mismo usaré la instrucción *is*, de forma que puedo preguntar si un objeto es la misma cosa que otro objeto:

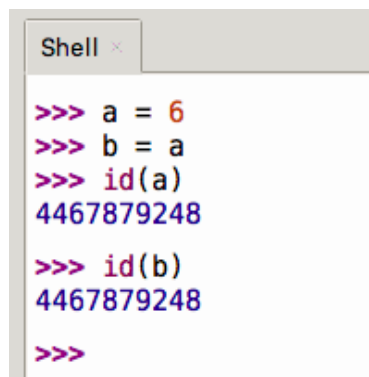


```
Shell x
>>> a = 6
>>> b = 6
>>> id(a)
4467879248
>>> id(b)
4467879248
>>> a is b
True
>>>
```

¿Ves? Al preguntar si *a* es *b* me contesta que es cierto (*True*). A priori podríamos decir que tenemos dos variables, *a* y *b*, y que contienen el mismo valor pero no son la misma cosa... pero es que en Python actualmente sí son la misma cosa, porque ambos nombres de las variables apuntan a la misma dirección de memoria.

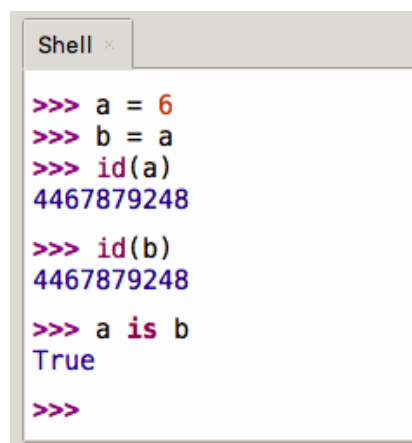
Todo esto puede resultar confuso, de hecho al empezar en Python programadores avezados en otros lenguajes tienen que investigar bien cómo declara y almacena las variables Python.

Podemos también copiar variables y ver que ocurre lo mismo:



```
Shell x
>>> a = 6
>>> b = a
>>> id(a)
4467879248
>>> id(b)
4467879248
>>>
```

La variable *a* vale 6, y posteriormente creamos una variable *b* a la cual asignamos el valor de *a*, por lo que ambas apuntan a la misma dirección de memoria, tienen la misma identidad. Estamos diciendo básicamente que *a* y *b* son la misma cosa.



```
Shell x
>>> a = 6
>>> b = a
>>> id(a)
4467879248
>>> id(b)
4467879248
>>> a is b
True
>>>
```

Por lo tanto podemos decir que si dos variables contienen el mismo elemento también son la misma cosa:

```
Shell x
>>> a = 6
>>> b = 6
>>> a == b
True
>>> a is b
True
>>> |
```

Hay otro tipo de variable en Python llamado **lista** que sí es **mutable**. Más adelante trabajaremos con ella, pero sólo para que veas la diferencia, vamos a declarar dos listas iguales y ver si son el mismo objeto:

```
Shell x
>>> a = [2,3,5]
>>> b = [2,3,5]
>>> a is b
False
>>> |
```

Vemos que la lista es básicamente lo que su propio nombre dice, una lista de cosas. En nuestro caso tenemos dos listas y ambas contienen el número 2, el número 3 y el número 5, pero Python nos dice que no son la misma cosa. Veamos su identidad:

```
Shell x
>>> a = [2,3,5]
>>> b = [2,3,5]
>>> a is b
False
>>> id(a)
4478146248
>>> id(b)
4480254088
>>> |
```

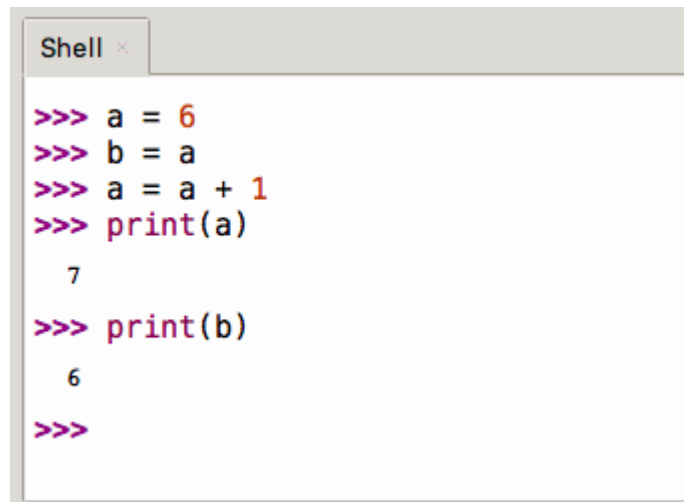
Efectivamente, a no apunta al mismo sitio que b, es decir, su identidad o dirección en memoria es diferente, por lo que no son la misma cosa, aunque sí que tienen el mismo valor:

```
Shell x
>>> a = [2,3,5]
>>> b = [2,3,5]
>>> a is b
False
>>> a == b
True
>>>
```

El resumen de todo esto es que en Python las variables son **no mutables**, cosa bastante irónica, y por lo tanto cada vez que le damos un nuevo valor a una variable estamos realmente creando una nueva variable con el mismo nombre en una nueva ubicación de memoria... ¡¡¡que follón!!!

Todo lo anterior no es determinante para aprender a programar, y si no has programado en ningún lenguaje puede que te preguntes por qué es importante, pero no se puede aprender a programar sin comprender cómo funciona la declaración de diferentes objetos, como las variables, y dónde se almacenan.

Sabiendo que Python crea una nueva ubicación para una variable al cambiar el valor de la misma y, por lo tanto, está creando una nueva variable a todos los efectos, podemos ver el siguiente ejemplo:

A screenshot of a Python Shell window. The window has a title bar with the text 'Shell' and a close button. The shell contains the following code and output:

```
>>> a = 6
>>> b = a
>>> a = a + 1
>>> print(a)
7
>>> print(b)
6
>>>
```

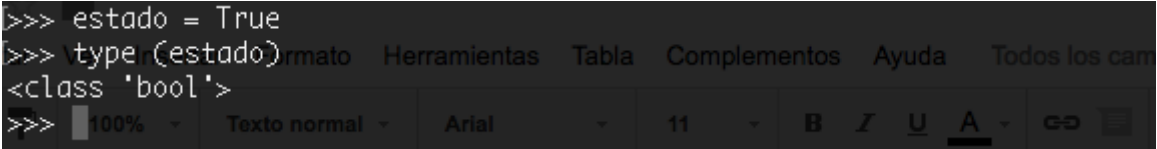
Al principio la variable *a* vale 6 y la variable *b* vale lo mismo que la variable *a*. Esto quiere decir que apuntan al mismo lugar de la memoria. Al indicar que la variable *a* pasa a contener el valor que contenía incrementado en 1, es decir, 7, Python crea un nuevo lugar en memoria para la variable *a* y le asigna el valor 7. En cambio la variable *b* sigue valiendo 6.

Si las variables fuesen mutables y pudiésemos cambiar su valor, al cambiar el valor de *a* y almacenarlo en la posición que ocupaba su anterior valor, también cambiaría *b*. Y esto es exactamente lo que ocurre en muchos otros lenguajes de programación, donde al cambiar una variable de valor estás cambiando todas aquellas que apuntan a la misma dirección de memoria, cosa que no ocurre en Python. Es una gran ventaja con un coste pequeño, el de decir que las variables son no mutables, lo cual es un problema sintáctico, ¡pero ahorra mucho sufrimiento programando!

4.1 True y False

Ahora que hemos conocido los valores *True* y *False* podemos conocer un nuevo tipo de variable, la **variable booleana**, que sólo tiene dos estados (*True* y *False*):

```
>>> estado = True
>>> type(estado)
<class 'bool'>
```

A screenshot of a Python interpreter window. The window has a dark background with light-colored text. The code entered is: >>> estado = True, followed by >>> type(estado). The output is <class 'bool'>. The window title bar shows 'rmato', 'Herramientas', 'Tabla', 'Complementos', 'Ayuda', and 'Todos los cam'. The status bar at the bottom shows '100%', 'Texto normal', 'Arial', '11', and various formatting icons like Bold, Italic, Underline, and Align.

Este tipo de variables se usan para todo aquello que sólo puede tomar dos estados (que en programación es algo habitual).

Es una variable muy útil en programación y, además, ocupa muy poca memoria, pues necesita sólo almacenar un 0 (False) o un 1 (True). La vamos a usar en muchas ocasiones.

5. CONDICIONALES

Pasemos ahora a ver una nueva funcionalidad de Python (y de cualquier otro lenguaje de programación), el uso de condicionales.

El condicional es un elemento de control que se escribe como *if* (en castellano si). El condicional *if* nos permite, al ejecutarse, realizar una acción sólo si se cumple una o varias condiciones. Por ejemplo: si el número es par haz tal cosa, si es impar haz tal otra y si no es par ni impar haz esta otra cosa.

Vamos directamente a ver y probar un código que contiene un *if* para entender cómo trabaja:

```
numero = input("Elige un número: ")
try:
    numero = int(numero)
except:
    numero = input("No has elegido un número, elige un número: ")
    numero = int(numero)
if numero % 2 == 0:
    print("Has elegido un número par")
```

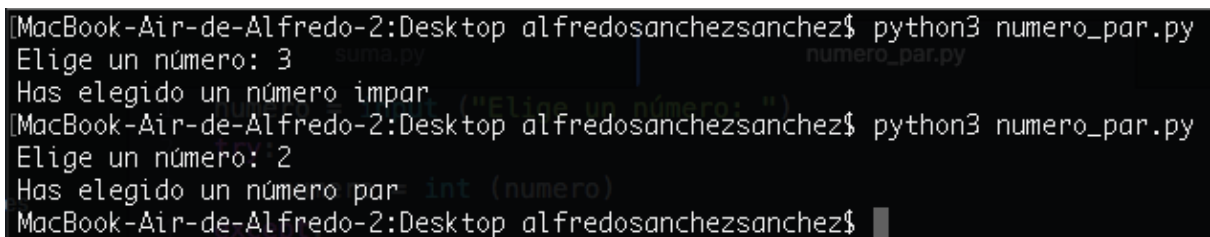
Prueba el código anterior teniendo cuidado con las indentaciones. Guárdalo en un script y ejecútalo para ver qué ocurre cuando eliges un número. No sigas leyendo hasta que no lo hayas probado.

Un *if* requiere comprobar una condición para ejecutarse. En nuestro ejemplo hemos introducido un número y el programa comprueba cual es el resto de la división de ese número entre 2 (*numero % 2*) y lo compara con el valor 0.

Al ser *numero* una variable de tipo entero el resultado de dividirla entre 2 no puede ser otro que 0 o 1. Si queremos que nos diga, así mismo, si el número elegido es impar en caso de ocurrir podemos añadir un segundo *if*:


```
numero = input("Elige un número: ")
try:
    numero = int(numero)
except:
    numero = input("No has elegido un número, elige un número: ")
    numero = int(numero)
if numero % 2 == 0:
    print("Has elegido un número par")
if numero % 2 == 1:
    print("Has elegido un número impar")
```

Aquí tienes un par de pruebas con el código anterior:



```
MacBook-Air-de-Alfredo-2:Desktop alfredosanchezsanchez$ python3 numero_par.py
Elige un número: 3
Has elegido un número impar
MacBook-Air-de-Alfredo-2:Desktop alfredosanchezsanchez$ python3 numero_par.py
Elige un número: 2
Has elegido un número par
MacBook-Air-de-Alfredo-2:Desktop alfredosanchezsanchez$
```

Como ves, al ejecutar el programa puedo probar números pares e impares y el resultado es el esperado.

El problema es que a veces tenemos un montón de posibilidades a comprobar y no es buena práctica incluir cada una de ellas en un *if* diferente porque Python tendrá que comprobar todas ellas una por una aunque la primera ya haya sido la elegida. En nuestro ejemplo, sin ir más lejos, Python comprueba si el número es impar aunque ya haya certificado que es par. Imagina que estamos comprobando algo más complejo, como la talla de camiseta de una persona. Si mide tanto, dale una XS, si mide tanto, dale una S, si mide tanto dale una M... y así hasta la XXXL. El programa iría comprobando cada *if*, independientemente de que el primero ya se haya cumplido.

Aquí nos encontramos con una de las grandes diferencias entre el lenguaje humano y el lenguaje máquina. Un humano sabría que si ya hemos decidido que su talla es la XS no tiene por qué comprobar si las demás tallas son su talla, total, ¡ya lo hemos decidido! En cambio una máquina necesita saber explícitamente que no tiene que comprobar nada más una vez que ya una de las condiciones se cumple.

Para ello, existe una instrucción ampliada del condicional *if* que nos permite ir chequeando diferentes comparativas hasta que una se cumpla. Para cada nuevo valor que quiera chequear y esté vinculado al primer *if* añadiré el código *else if* o *elif*. Esta instrucción toma la siguiente forma:

```
if numero == 0:
    print("Has elegido el 0")
elif numero == 1:
    print("Has elegido el 1")
elif numero == 2:
    print("Has elegido el 2")
...
```

Podríamos seguir poniendo tantas comprobaciones como necesitáramos pero una vez que, al ejecutarse el programa, una sea cierta, el resto no se comprobarán. Esto hace que los programas sean bastante más ágiles y rápidos pues en ocasiones hay que realizar muchas comparaciones. En el ejemplo de las camisetas sería: comprueba si su talla es la XS, si no es esa, comprueba si es la S, si no es esa comprueba si es la M... En el momento que una sea la talla adecuada deja de comprobar, ya se ha cumplido una de las condiciones a comprobar.

Existe una tercera opción además del *if* y *elif*, sólo podemos usarla al final de nuestra lista de chequeos o comprobaciones y se aplica a cualquier otra condición no incluida en las comprobaciones hechas. Dicha opción será *else* a secas y, en caso de existir, su contenido siempre se ejecuta si no se ha cumplido ninguna de las anteriores comprobaciones.

Por seguir con el ejemplo, si mide tanto dale una XL, si mide tanto dale una XXL, y si no mide nada de lo dicho hasta ahora, dale una XXXL (y que se conforme, que no hay más opción).

Prueba el siguiente código guardándolo en un script .py y elige diferentes números para ver cómo funciona:

```
numero = input("Elige un número del 0 al 2: ")
if numero == "0":
    print("Has elegido el 0")
elif numero == "1":
    print("Has elegido el 1")
elif numero == "2":
    print("Has elegido el 2")
else:
    print("El número elegido no está entre el 0 y el 2 o no es un número")
```

Como podrás ver, he hecho la comprobación con un número entre comillas. En esta ocasión en vez de cambiar el contenido de la variable *numero* a *int* directamente compruebo en los condicionales con un número como *string*, que es lo que almacena un input.

Aquí tienes el resultado de probarlo con varios números:

```
MacBook-Air-de-Alfredo-2:Desktop alfredosanchezsanchez$ python3 condicional.py
Elige un número del 0 al 2: 1
Has elegido el 1
MacBook-Air-de-Alfredo-2:Desktop alfredosanchezsanchez$ python3 condicional.py
Elige un número del 0 al 2: 2
Has elegido el 2
MacBook-Air-de-Alfredo-2:Desktop alfredosanchezsanchez$ python3 condicional.py
Elige un número del 0 al 2: 3
El número elegido no está entre el 0 y el 3 o no es un número
```

A los condicionales que incluyen varias condiciones se les llama *if - else*.

Ahora que ya conoces el condicional vamos a empezar a hacer un programa funcional. Vamos a crear un script donde se pregunte al usuario por dos números y se le dé varias opciones para seleccionar:

```
numero_1 = input("Elige el primer número: ")
try:
    numero_1 = int(numero_1)
except:
    numero_1 = input("No has elegido un número, elige el primer número: ")
    numero_1 = int(numero_1)
numero_2 = input("Elige el segundo número: ")
try:
    numero_2 = int(numero_2)
except:
    numero_2 = input("No has elegido un número, elige el segundo número: ")
    numero_2 = int(numero_2)
print("Puedes elegir los siguientes números:")
print("1 - Sumar ambos números")
print("2 - Restar ambos números")
seleccion = input("¿Cual es tu selección?: ")
```

Prueba a realizar tú mismo el siguiente ejercicio antes de seguir con el documento. Intenta continuarlo incluyendo a continuación del código un condicional que sume ambos números si la selección es 1, que reste ambos números si la selección es 2 y que informe de que la selección es incorrecta en caso de no ser 1 ni 2. Es importante que lo intentes antes de ver cómo se haría.

Si no lo consigues puedes seguir leyendo:

```
numero_1 = input("Elige el primer número: ")
try:
    numero_1 = int(numero_1)
except:
    numero_1 = input("No has elegido un número, elige el primer número: ")
numero_2 = input("Elige el segundo número: ")
try:
    numero_2 = int(numero_2)
except:
    numero_2 = input("No has elegido un número, elige el segundo número: ")
print("Puedes elegir los siguientes números:")
print("1 - Sumar ambos números")
print("2 - Restar ambos números")
seleccion = input("¿Cual es tu selección?: ")
if seleccion == "1":
    print ("Resultado de la suma:")
    print (numero_1 + numero_2)
elif seleccion == "2":
    print ("Resultado de la resta:")
    print (numero_1 - numero_2)
else:
    print ("No has elegido una opción válida")
```

Para que el contenido del *if - else* se muestre impreso en una sola línea (en nuestro ejemplo está en dos líneas) deberíamos añadir el número resultado de la operación al *string* “Resultado de la...” y para ello previamente deberíamos guardar el resultado de la operación en una variable y convertirlo a *string* o bien concatenar la operación al texto con una coma. Sería importante, llegados a este punto, que entiendas de qué estamos hablando y sepas hacerlo (aunque te cueste revisar unas cuantas cosas del tema anterior). También podrías concatenar la frase y el resultado de la suma usando la coma.

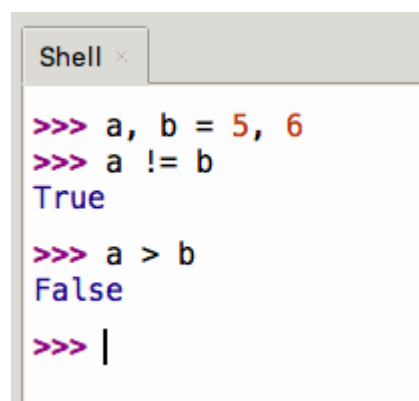
Ahora que ya sabes hacer tu primer script con condicionales intenta añadir dos nuevas opciones para multiplicar los números y dividirlos. Debes incluirlas como opción dada al usuario (que sepa que puede elegir las) y añadir los *if - else* necesarios para que se ejecuten esas opciones si las elige el usuario.

5.1 ¿Sólo puedo comprobar igualdad?

Existen más instrucciones para comprobar dos variables u objetos, las dejo anotadas en modo código, son importantes:

```
a != b #Comprobamos si a es distinto de b
a < b #Comprobamos si a es menor a b
a > b #Comprobamos si a es mayor a b
a <= b #Comprobamos si a es menor o igual a b
a >= b #Comprobamos si a es mayor o igual a b
```

Puedes declarar dos variables numéricas a y b y hacer comprobaciones de las anteriormente descritas para ver el efecto, como en la siguiente imagen:



```
Shell x
>>> a, b = 5, 6
>>> a != b
True
>>> a > b
False
>>> |
```

¡Recuerda que puedes declarar tantas variables simultáneas como quieras!

6. OPERADORES LÓGICOS (AND Y OR)

Imagina que al ejemplo de la talla sumamos otra comprobación. Si mide tanto y es verano, dale una camiseta talla M. Tenemos ya dos cosas a comprobar.

Una forma de hacerlo sería anidando un condicional dentro de otro, como en el siguiente código para comprobar que un valor sea mayor o menor que un número y a la vez comprobar si es par o impar:

```
numero = int(input("Elige un número: "))
if numero%2 == 0:
    #Si hemos entrado es porque el número es par
    if numero > 9:
        print("El número elegido es par y mayor a 9")
    else:
        print("El número elegido es par y menor o igual a 9")
else:
    #Si hemos entrado es porque el número no es par
    if numero > 9:
        print("El número elegido es impar y mayor a 9")
    else:
        print("El número elegido es impar y menor o igual a 9")
```

Anidar condicionales es una forma de solucionar el problema, pero no sólo existe la posibilidad de comprobar una única igualdad en cada condicional, sino que podemos utilizar dos operadores lógicos para que nuestro condicional haga más de una comprobación.

Para este tipo de comprobaciones podemos usar el operador *and*. Copia y prueba el siguiente código para entenderlo mejor:

```
numero = int(input("Elige un número: "))
if numero > 9 and numero%2 == 0:
    print("El número es mayor que 9 y es par")
elif numero > 9 and numero%2 == 1:
    print("El número es mayor que 9 y es impar")
elif numero <= 9 and numero%2 == 0:
    print("El número es menor que 9 y es par")
else:
    print("El número es menor que 9 y es impar")
```

¿Ya lo has probado? Como verás, estamos comprobando dos cosas, la primera si el número es mayor que 9 o menor o igual a 9 y la segunda si es par o impar. Con dos comprobaciones binarias (es decir, cada una tiene dos posibilidades) tenemos un total de 4 posibilidades. De esta forma, podemos comprobar tres de ellas y sabemos que si no se han cumplido estaremos en la cuarta posibilidad.

En el ejemplo anterior no hemos aplicado ningún elemento de control de errores, por lo que si el usuario contesta algo que no sea un número... ¡error al canto!

Podríamos utilizar tantos *and* como quisiésemos, pudiendo hacer más de dos comprobaciones (tres, cuatro o las que sean necesarias), pero si tenemos que hacer muchas comprobaciones en un condicional es muy posible que haya un camino más sencillo, quizá una de las comprobaciones sea más determinante que otra y se pueda hacer primero o algo similar.

El otro operador lógico es el operador *or*, que quiere decir o, y sirve para realizar dos o más comparaciones y devuelve True si al menos una de las comparaciones es cierta.

Imagina la siguiente afirmación: si el terreno es blando o tiene una capa de materia orgánica profunda la cimentación se hará mediante pilotes. Con la afirmación anterior deberemos usar pilotes para cimentar si el terreno es blando *or* si el terreno tiene una gruesa capa de materia orgánica *or* si ocurren ambas cosas a la vez.

Como puedes ver, el operador *or* incluye el operador *and* entre una de las opciones en las que se cumple.

El siguiente código sirve a modo de ejemplo de cómo usar un operador *or*.

```
numero = int(input("Elige un número: "))
if numero > 9 or numero%2 == 0:
    print("El número es mayor que 9 o es par o ambas cosas")
else:
    print("El número no es mayor que 9 ni es par")
```

El programa entrará en el condicional *if* si el número es mayor que 9 o bien si el resto del número dividido entre 2 es cero (es par) o bien si ocurren ambas cosas. En caso de no ocurrir ninguna de las dos cosas, entrará en el *else*.

7. BUCLE WHILE

Una vez que sabemos cómo alterar la ejecución de un programa en función de una decisión puntual, vamos a ver otro tipo de elemento de control, en este caso uno que se repetirá mientras (*while*) o hasta que se cumpla una condición.

Volvamos a nuestro programa *try - except* inicial:

```
print ("Vamos a sumar dos números")
numero_1 = input("¿Cuál es el primer número?: ")
numero_2 = input("¿Cuál es el segundo número?: ")
try:
    numero_1 = int(numero_1)
except:
    numero_1 = input("No has elegido un número, elige un número: ")
    numero_1 = int(numero_1)
numero_2 = int(numero_2)
suma = numero_1 + numero_2
suma = str(suma)
print("La suma de ambos números es " + suma)
```

En dicho programa faltaba implementar la solución para el número 2, más adelante tendrás que completarlo si no lo hiciste en su momento.

¿Qué tal si forzamos al usuario a que la variable que introduzca en el input *numero_1* sea un número repitiendo una y otra vez la pregunta hasta que se pueda convertir en *int* la respuesta?

Para este tipo de acciones (y muchas otras) se usan los bucles. Vamos a aprender a usar el bucle *while* (mientras). Dicho bucle se repetirá mientras se cumpla una condición, y también vamos a aprender a utilizar las variables como elementos para provocar o dejar de provocar bucles.

Para ello, usaremos variables booleanas (recuerda, las que pueden tomar el valor *True* o *False*).

La forma de entender este proceso es sencilla:

1. Declaramos una variable con el valor *True*.
2. Creamos un bucle *while* que se repita hasta que la variable declarada tenga el valor *False*.
3. Dentro del bucle situamos el código que necesitemos que se repita hasta la solución correcta. Dicho código puede tener un *try - except* en el cual dentro del *try* situemos el cambio de la variable booleana de *True* a *False*.

Veámoslo con un código sencillo:

```
noEsNumero = True
numero_1 = input("¿Cuál es el primer número?: ")
while noEsNumero == True:
    try:
        numero_1 = int(numero_1)
        noEsNumero = False
    except:
        numero_1 = input("No has elegido un número, elige un número: ")
print("El número elegido es", numero_1)
```

Aquí tienes una captura de la ejecución del mismo:

```
prueba4.py x
1 noEsNumero = True
2 numero_1 = input("¿Cuál es el primer número?: ")
3 while noEsNumero == True:
4     try:
5         numero_1 = int(numero_1)
6         noEsNumero = False
7     except:
8         numero_1 = input("No has elegido un número, elige un número: ")
9 print("El número elegido es", numero_1)

Shell x
>>> %Run prueba4.py
¿Cuál es el primer número?: a
No has elegido un número, elige un número: x
No has elegido un número, elige un número: r
No has elegido un número, elige un número: 5
El número elegido es 5
>>>
```

Como ves, el usuario no tiene más remedio que poner un número ya que se repite la pregunta **hasta** que así sea o, dicho de otra forma, **mientras** que no sea así.

Verás que la cosa se va complicando, ya estamos usando un bucle y un *try - except* simultáneamente. Empieza a ser un problema no entender qué hace el programa. En tu código puedes ir poniendo comentarios para explicarlo, tratar de explicar un código es una buena práctica para entender perfectamente qué hace.

Lo más importante es entender **cómo usamos una variable para entrar y salir de un bucle** y cómo usamos el bucle para forzar al usuario a elegir un número. El cambio de la variable *noEsNumero* de *True* a *False* produce que se termine el bucle *while*. Ese cambio no ocurre hasta después de que el programa pueda convertir *numero_1* en entero, y si este proceso da error pasará directamente al *except* y se saltará el cambio de la variable *noEsNumero* de *True* a *False*.

El orden dentro del *try* es importante, puesto que Python va a pasar del *try* al *except* en el momento que algo del *try* de error, pero todo lo que haya hecho hasta ese momento queda hecho. Fíjate en el siguiente código:

```
numero = input("Dí un número: ")
noEsNumero = True
while noEsNumero == True:
    try:
        noEsNumero = False
        numero = int(numero)
    except:
        numero = input("No has elegido un número, dí un número: ")
```

Si te fijas he cambiado el orden dentro del *try*, primero sitúo el cambio de variable a *False* y luego ya intento convertir en entero la variable *numero*. Si el usuario contesta mal el error va a aparecer en la segunda línea del *try*, y la primera (el cambio de variable booleana) ya se habrá producido. Al dar error en el *try* se ejecutará el *except*, pero tras terminar el *except* el programa volverá a comprobar si se cumple la condición del bucle *while* y ésta ya no se estará cumpliendo, por lo que no se volverá a ejecutar su contenido. Aquí tienes una captura de la ejecución del código anterior:

```
prueba.py x
1  numero = input("Dí un número: ")
2  noEsNumero = True
3  while noEsNumero == True:
4      try:
5          noEsNumero = False
6          numero = int(numero)
7      except:
8          numero = input("No has elegido

Shell x
>>> %Run prueba.py
    Dí un número: A
    No has elegido un número, dí un número: A
>>> |
```

Como verás, no se repite el bucle hasta que el *input* sea un número porque la variable que rompe el bucle está mal situada antes de la comprobación del *try* que lanzará el *except*. Podemos resumir diciendo que realmente el *except* es lo que hará en el momento que algo del *try* de error, pero hasta ese punto puede haber hecho bastantes cosas del *try*.

Intenta, ahora que has visto este programa, repetirlo para seleccionar un segundo número y terminar preguntando al usuario qué quiere hacer con esos números (sumarlos, restarlos, multiplicarlos o dividirlos...). En este punto has llegado a un paso importante que todo programador acaba dando: **reutilizar código**. Utiliza el programa del apartado Condicionales para agilizar el proceso y completar el reto lanzado.

8. INTRODUCCIÓN A LAS LIBRERÍAS

Vamos a utilizar el bucle *while* para otra función, aprendiendo a la vez a usar bibliotecas, más conocidas comúnmente como librerías.

Una librería o biblioteca (del inglés *library*) es un código o conjunto de códigos funcionales que podemos utilizar importándolos a nuestro programa. Dicho de otra forma, son programas que nos permiten ser utilizados simplemente cargándolos y utilizándolos enteros o algunas partes del mismo. Es muy probable que veas más veces la palabra librería que la palabra biblioteca para referirse a esto, aunque es una mala traducción del inglés.

Al cargar una librería el código que contiene queda oculto, pero el programa lo incorpora para ser usado, aunque en nuestro código sólo aparezca la indicación de haber incorporado la librería.

Las librerías pueden tener muchos orígenes pero siempre hay información sobre su uso y opciones en internet. Cualquier persona que crea una biblioteca suele disponer en la red las instrucciones concisas de todas sus funcionalidades y posibilidades. Hacia el final del curso trabajaremos algunas de las bibliotecas más famosas y conocidas y veremos cómo aprender a usarlas a través de la información publicada en la red.

Veamos un ejemplo con la biblioteca *time*.

La biblioteca o librería *time* nos permite usar varias funciones, entre ellas establecer tiempos de espera en nuestro programa antes de ejecutar la siguiente línea de código. Vamos a hacer una cuenta atrás con las siguientes instrucciones:

- El usuario debe seleccionar el tiempo desde el que quiere cronometrar.
- Cuando lleguemos a cero pararemos la cuenta atrás.
- El tiempo disminuirá de segundo en segundo.

Para utilizar una librería debemos usar el siguiente código:

```
import time
```

En ocasiones usaremos una parte de una librería y no entera, lo veremos más adelante. La opción anterior importa a tu programa **toda** la librería *time*.

Ahora vamos a crear un programa sencillo y básico y luego propondré mejoras para que puedas practicar. Observa el siguiente código:

```
import time
tiempo = input("¿Cuánto tiempo tendrá la cuenta atrás?: ")
tiempo = int(tiempo)
while tiempo >= 0:
    print(tiempo)
    tiempo = tiempo - 1
    time.sleep(1)
```

Prueba a guardarlo como script y ejecutarlo, elige un tiempo prudencial (10, por ejemplo) y mira qué ocurre antes de seguir.

Si ya lo has probado deberías haber conseguido algo similar a esto:

```
MacBook-Air-de-Alfredo-2:Desktop alfredosanchezsanchez$ python3 cronometro.py
¿Cuánto tiempo tendrá la cuenta atrás?: 10
10
9
8
7
6
5
4
3
2
1
0
```


Si observas el código verás que hemos usado dos instrucciones nuevas. En primer lugar hemos usado una librería importándola y ejecutando una de sus funciones. Las funciones de las librerías se ejecutan con el nombre de la librería seguido de un punto y el nombre de la función que necesitamos, terminando siempre con una apertura de paréntesis y un cierre de paréntesis. Algunas funciones de las librerías necesitan, además, parámetros, que irán entre esos paréntesis.

En nuestro caso la función `time.sleep()` necesita entre paréntesis el tiempo (en segundos) que va a esperar el programa antes de continuar.

También se puede incorporar una librería asignándole un nuevo nombre. En nuestro ejemplo `time` no es un nombre muy extenso, pero podríamos importarla llamándola únicamente `tm` y cambiando `time` por `tm` en el programa. A continuación tienes el cambio incorporado al programa y su ejecución:

```
prueba.py x
1 import time as tm
2 tiempo = input("¿Cuánto tiempo tendrá la cuenta atrás?: ")
3 tiempo = int(tiempo)
4 while tiempo >= 0:
5     print(tiempo)
6     tiempo = tiempo - 1
7     tm.sleep(1)

Shell x
>>> %Run prueba.py
¿Cuánto tiempo tendrá la cuenta atrás?: 3
3
2
1
0
>>> |
```

Veremos más funciones de las librerías en otros temas, pero de momento podemos quedarnos con esta función tan interesante: ¡esperar una cantidad de tiempo!

Además, hemos utilizado una nueva forma de comparación en el *while*. No hemos usado un `==` sino un `>=`. Esta opción, ubicada en el código *tiempo* `>= 0` indica al programa que la comparación será verdadera cuando *tiempo* tenga un valor mayor o igual a cero. Al estar dentro de un *while* ese bucle se repetirá hasta que *tiempo* tenga un valor inferior a 0, de ahí que la cuenta atrás se detenga al llegar a cero.

Veamos un pequeño cambio que puede suponer una gran diferencia para entender el bucle. Efectúa el cambio en el código que verás a continuación y pruébalo como script:

```
import time
tiempo = input("¿Cuánto tiempo tendrá la cuenta atrás?: ")
tiempo = int(tiempo)
while tiempo >= 0:
    tiempo = tiempo - 1
    print(tiempo)
    time.sleep(1)
```

Como puedes observar, el único cambio es situar primero el cambio de la variable *tiempo* y después imprimimos la variable *tiempo*.

El resultado al probarlo debería ser algo similar a esto (he elegido 5 como *input* para reducir el tiempo de espera):

```
[MacBook-Air-de-Alfredo-2:Desktop alfredosanchezsanchez$ python3 cronometro.py
¿Cuánto tiempo tendrá la cuenta atrás?: 5
4
3
2
1
0
-1
```

Ese pequeño cambio resulta en un programa erróneo. Es importante pensar en **cómo las fases del programa se suceden** y qué necesito hacer primero. Si yo quiero hacer una cuenta atrás de 5 segundos, primero tendré que imprimir el 5, después lo convierto en un 4 y tras una espera de un segundo lo imprimo. Podría primero esperar un segundo y después convertirlo en 4 e imprimirlo, pero si primero le quito una unidad y luego lo imprimo mi cinco inicial se convertirá primero en un 4 y luego lo imprimiré, lo cual no es lo que queremos.

Así mismo, el programa termina en un -1. Esto es así porque (y esto es importante) la condición que contiene el *while* se comprueba tras cada ejecución de su contenido **completo**, es decir, el programa realiza todo el contenido del *while* en cada ejecución del mismo antes de comprobar si la condición sigue siendo válida. Para entender esto mejor vamos a poner un ejemplo muy claro:

```
import time
tiempo = input("¿Cuánto tiempo tendrá la cuenta atrás?: ")
tiempo = int(tiempo)
while tiempo >= 0:
    print(tiempo)
    tiempo = tiempo - 1
    time.sleep(1)
print("La variable tiempo ha acabado con un valor de: ")
print(tiempo)
```

Si ejecutamos este código eligiendo como *input* 5 segundos obtendremos lo siguiente:

```
[MacBook-Air-de-Alfredo-2:Desktop alfredosanchezsanchez$ python3 cronometro.py
¿Cuánto tiempo tendrá la cuenta atrás?: 5
5
4
3
2
1
0
La variable tiempo ha acabado con un valor de:
-1
```

Es importante entender por qué la variable ha acabado con un valor de -1. El programa imprime la variable *tiempo* antes de realizar ninguna otra acción del *while*. En la última ejecución del *while* el programa imprime el 0, después le resta 1 y la variable *tiempo* pasa a tener el valor -1 y posteriormente espera 1 segundo. Tras ello, salimos del *while* al volver a comprobar si *tiempo* es mayor o igual que 0 y, ya fuera del *while*, imprimimos de nuevo el valor de la variable *tiempo*, que tiene el valor final de -1.

Seguramente en el futuro cambiarás muchas cosas de este programa, pero en este punto es suficiente para nuestra cuenta atrás. Sólo añadiré que realmente no está pasando un segundo exacto en cada ejecución del *while*, pues el resto de acciones también lleva un tiempo, aunque sea muy pequeño, pero a efectos prácticos tenemos una cuenta atrás en segundos.

El reto, ahora, es que consigas hacer una cuenta atrás que pregunte el tiempo al usuario y lo repita hasta que introduzca un número correcto y que, tras llegar a cero, imprima el valor “Cuenta atrás terminada”.

9. BUCLE FOR

Hemos visto un tipo de bucle que se va a repetir hasta que se deje de cumplir una condición o, dicho de otra forma, se va a repetir mientras se cumpla una condición, lo cual es muy útil si la repetición es fruto de una condición, pero hay ocasiones que sólo queremos repetir un número de veces, sin necesidad de cumplirse una condición. Por ejemplo, si queremos hacer una cuenta atrás desde 10 hasta 0, sabemos que tenemos que repetir la instrucción un número concreto de veces, y si siempre va a ser desde 10 no necesitamos comprobar nada, lo repetimos 11 veces (pues incluye el 0) y listos.

Para repeticiones dictadas por un número y no por una condición usaremos el bucle *for*. Este bucle es muy usado para repetir acciones cuando conocemos el número de veces que se va a repetir, es decir, para aquellos bucles en los cuales sabemos cuando va a comenzar y cuando va a terminar.

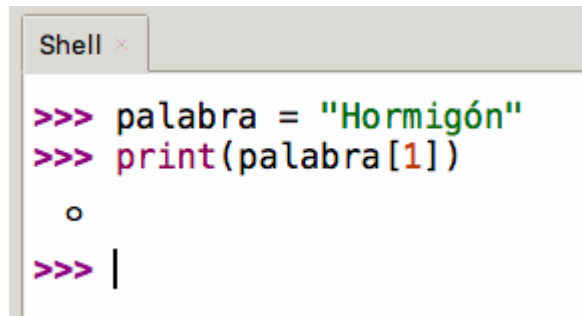
Antes de introducirnos en el bucle vamos a ver un ejemplo con un *while* y después lo trasladaremos a un *for*.

Imagina que tenemos que escribir todas las letras que componen una palabra. Para ello tenemos que ir recorriendo la palabra e imprimiendo la letra que ocupa cada posición de la misma.

Aprendamos algunas cosas nuevas sobre los *string* (en castellano, cadenas). ¿Cómo se nombra la posición que ocupa cada carácter de un *string*? Prueba el siguiente código en tu terminal o Thonny:

```
palabra = "Hormigón"  
print(palabra[1])
```

Analizando el código podemos intuir que imprimiremos la posición 1 del *string* almacenado en la variable *palabra*. Ahora bien, si lo probáis veréis lo siguiente:

A screenshot of a Python Shell window. The window has a title bar that says "Shell" with a close button. Inside the shell, the following code is entered and executed:

```
>>> palabra = "Hormigón"
>>> print(palabra[1])
o
>>> |
```

Resulta que la posición 1 de la palabra Python la ocupa la **o**, cuando lo lógico para un ser humano hubiese sido pensar que la posición 1 sería la **H**. Pues bien, en lenguaje máquina **siempre empezaremos a numerar y contar desde cero**. Esto es muy importante y al principio cuesta un poco pensar en numerar desde cero, pero con la práctica lo naturalizarás.

Si pensamos un poco en cómo contamos los humanos hay cosas que si empezamos a contar desde cero, por ejemplo los años de vida, nuestro primer año de vida es el cero. Hay otras que empezamos en el uno, por ejemplo las cantidades de productos que tenemos.

Siguiendo con nuestras cadenas, es muy útil saber el número de caracteres que tiene una cadena. Veamos cómo se podría realizar esta acción:

```
palabra = "Hormigón"
print(len(palabra))
```

Sencillamente debemos indicar que queremos conocer la longitud de un string con el comando *len* seguido del nombre de la variable donde está el string.

Aquí tienes la ejecución del código anterior:

```
>>> palabra = "Hormigón"
>>> print(len(palabra))
8
>>> |
```

También se puede utilizar directamente para medir un string no almacenado:

```
print(len("Hormigón"))
```

Una vez que ya sabemos cómo conocer la posición de un carácter en una cadena y cómo medir su longitud vamos a generar un código que deletree una palabra.

Analiza, guarda y prueba el siguiente código para intentar entender cómo funciona:

```
palabra = input("¿Qué palabra quieres deletrear?: ")
longitud = len(palabra)
posicion = 0
while posicion + 1 <= longitud:
    print(palabra[posicion])
    posicion = posicion + 1
```

Aquí tienes el resultado de su ejecución con la palabra “Python”:

```
[MacBook-Air-de-Alfredo-2:Desktop alfredosanchezsanchez$ python3 deletreo.py
¿Qué palabra quieres deletrear?: Python
P
y
t
h
o
n
```

Quizá no tengas muy claro por qué funciona el código, es algo habitual al ver código creados y es un momento estupendo para aprender a poner **comentarios**. Aunque ya los hemos mencionado antes, volvemos a explicarlo porque no viene nada mal. Los comentarios son textos que ponemos para explicar parte del programa y que Python no interpretará porque sabrá que son aclaraciones para el usuario. Los comentarios en Python se indican con una almohadilla si son de una línea:

#comentario

O bien con triples comillas si quiero hacer un comentario más largo:

“ “ “ comentario muy largo “ “ “

Veamos si ahora entiendes bien el código con esta captura de pantalla del editor Atom:

```
1  """Preguntamos al usuario qué palabra quiere deletrear"""
2  palabra = input ("¿Qué palabra quieres deletrear?: ")
3  """Almacenamos la longitud de la palabra en una variable"""
4  longitud = len (palabra)
5  """Creamos una variable para recorrer la palabra"""
6  posicion = 0
7  """Este bucle while se repetirá mientras el valor que tenga la variable
8  posición + 1 sea menor o igual que longitud. Hay que sumarle uno a la
9  variable posición porque empezamos a contar en 0 y la longitud de un
10 string cuenta el número de caracteres que tiene, por lo que una palabra
11 con una sola letra tendra una longitud de 1 y la posición de su letra
12 será la 0."""
13 while posicion + 1 <= longitud:
14     """Imprimimos el caracter que está en la variable posición"""
15     print (palabra[posicion])
16     """Sumamos uno a la variable posición para ir recorriendo el
17     string"""
18     posicion = posicion + 1
```

Si ya lo tienes un poco más claro podemos seguir. No tengas problema en expresar toda dificultad con las comparaciones entre valores que tienen diferentes rangos o que empiezan en diferentes números (unas en cero y otras en uno). El código anterior ya empieza a ser complejo y es normal tener dificultades para conseguir ver cómo funciona con claridad.

Éste es el caso de las posiciones de un string respecto de su longitud. Las posiciones empiezan en 0, la longitud empieza en 1, si queremos compararlas hay que conseguir que ambas empiecen en el mismo número, o bien en el 1 y habrá que sumar una unidad a la posición, o bien en el 0 y habrá que restar una unidad a la longitud.

Pasemos ahora a ver cómo funciona un bucle *for*. Prueba el siguiente código guardándolo en un *script*:

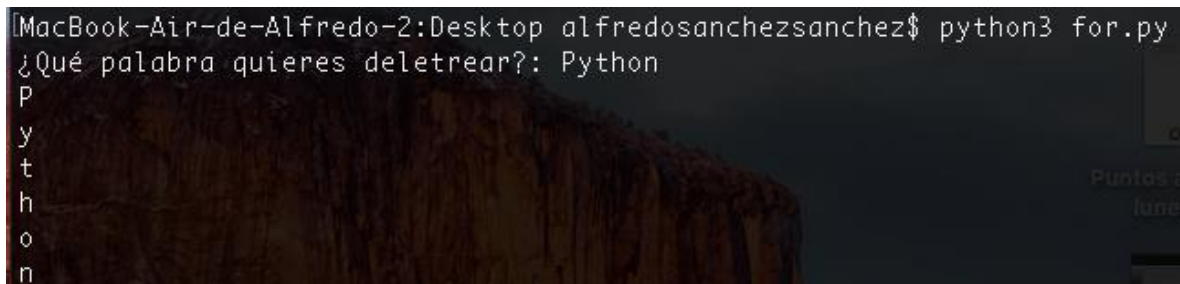
```
palabra = input("¿Qué palabra quieres deletrear?: ")
for letra in range(len(palabra)):
    print(palabra[letra])
```

Como habrás podido comprobar, es un deletreador perfecto y en bastantes menos líneas que el programa elaborado en un *while*. Un *for* es un bucle que se repite las veces que queramos, siguiendo un patrón que le indiquemos, por ejemplo, repite una acción contando hasta 10 de uno en uno o repite hasta 10 contando de dos en dos...

Los bucles *for* necesitan una **variable propia** para ir contando y almacenando el valor de la cuenta. Podemos declarar esa variable en el mismo *for*. En nuestro ejemplo, sin ir más lejos, la variable se llama *letra* y está **declarada directamente** en el bucle *for*. Habitualmente se usa la letra *i* para la variable propia del *for*.

Además, necesitan una indicación de cómo contar y hasta cuánto. En el ejemplo contaremos desde cero hasta la cantidad de caracteres que tenga el *string palabra*. Para ello hemos indicado que cuente dentro del rango (*range*) de la longitud de *palabra*.

Veamos un ejemplo de aplicación para explicarlo con más calma:



```
MacBook-Air-de-Alfredo-2:Desktop alfredosanchezsanchez$ python3 for.py
¿Qué palabra quieres deletrear?: Python
P
y
t
h
o
n
```

Hemos vuelto a deletrear la palabra *Python*, cuya longitud es 6, así que el bucle *for* viene a ser, traducido a nuestro lenguaje:

Para cada letra en el rango de 6 letras, repite:

Al ejecutarse el bucle *for* este va a ir contando hasta 6 y almacenando el valor de la cuenta en la variable *letra*, eso sí, empezando por 0 (como ya hemos dicho, en programación empezamos a contar en 0).

De esta forma, en la primera iteración o ejecución del bucle *for* la variable *letra* tiene el valor 0, por lo que si le pedimos que imprima la posición *letra* de la palabra *Python* imprimirá la P.

Pero, si le pido que cuente hasta 6 empezando en el 0, ¿no contará 7 veces? Es importante saber que el bucle *for*, al indicarle el rango en el que queremos contar siempre empezará en 0 y llegará hasta **el valor anterior al indicado**.

Si le decimos que cuente en un rango de 6 contará de la siguiente forma: 0, 1, 2, 3, 4 y 5. El valor indicado no lo incluye. Digamos que cuenta hasta el 6 pero el 6 no está incluido.

Como esta instrucción suele causar algún problema más en su comprensión vamos a desglosar el bucle tal como lo hace Python con la instrucción *for letra in range (len (palabra))* y *palabra = "Python"*

1. Comienza el bucle for. Para cada letra en el rango 6:
 - 1.1. Letra = 0
 - 1.1.1. Imprime la posición 0 de la variable Python: P
 - 1.1.2. Suma uno a letra, pasa a ser 1
 - 1.2. Letra = 1
 - 1.2.1. Imprime la posición 1 de la variable Python: y
 - 1.2.2. Suma uno a letra, pasa a ser 2
 - 1.3. Letra = 2
 - 1.3.1. Imprime la posición 2 de la variable Python: t
 - 1.3.2. Suma uno a letra, pasa a ser 3
 - 1.4. Letra = 3
 - 1.4.1. Imprime la posición 3 de la variable Python: h
 - 1.4.2. Suma uno a letra, pasa a ser 4

1.5. Letra = 4

1.5.1. Imprime la posición 4 de la variable Python: o

1.5.2. Suma uno a letra, pasa a ser 5

1.6. Letra = 5

1.6.1. Imprime la posición 5 de la variable Python: n

1.6.2. Suma uno a letra, pasa a ser 6

2. Fin del bucle

En este caso nos da igual con qué valor acabe la variable *Letra*.

Que código más efectivo y breve, ¿verdad? Pues aún se puede hacer más corto.

El bucle *for* admite muchas formas como instrucción de búsqueda. Pongamos algunos ejemplos:

```
for letra in range [0, 1, 2, 3, 4, 5]:  
    print (palabra [letra] )  
for letra in "Python":  
    print (letra)  
for letra in palabra:  
    print (letra)
```

Los tres bucles hacen exactamente lo mismo. En el primero están enumeradas las posiciones de cada letra de la palabra Python (cosa que no tiene mucho sentido a no ser que sepamos qué palabra concreta vamos a deletrear). El segundo deletrea el *string* que situemos en la condición del *for*, cosa que nos obliga a ceñirnos a un único string, y el tercero recorre la variable *palabra* e imprime cada carácter que contenga (al cual llamamos *letra*). El último caso es el más habitual, ya que recorreremos elementos que estén guardados en variables.

No vamos a hablar mucho más por ahora de los bucles *for*, iremos utilizándolos a lo largo del curso en otras ocasiones. Ahora toca hacer una prueba: intenta reproducir la cuenta atrás que hicimos con un bucle *while* utilizando ahora un bucle *for* tras preguntar al usuario por la cantidad de la cuenta atrás.

10. INSTRUCCIONES DE CÓDIGO USADAS

```
#Intentar ejecutar un código y evitar un error
try:
    #código
except:
    #código
#Identidad de una variable
id(variable)
#Código que se ejecuta si se cumple una condición (condicional)
if comparacion:
    #código
elif comparacion:
    #código
else:
    #código
#Código que se repita mientras se cumpla una condición (while)
while comparacion:
    #código
#Código que se repita una serie de veces (for)
for repeticiones:
    #código
#Establecer un rango de datos
range(datos)
#Instrucciones de comparación de valores
#Igualdad
==
#Mayor que, menor que, mayor o igual que, menor o igual que
>
<
>=
<=
```

```
#Desigualdad
!=
#Importar biblioteca
import biblioteca
#Importar biblioteca cambiando nombre
import biblioteca as nuevo_nombre
#Usar elemento de biblioteca
elemento.biblioteca()
#Importar parte de una biblioteca
from biblioteca import parte
#Usar una posición de un string
string[posicion]
#Longitud de un string
len(string)
```

11. RETO

Realiza un programa que realice una cuenta atrás de una cantidad de tiempo (minutos y segundos) introducida por el usuario. Dicho programa deberá contar cuántos minutos quedan a no ser que quede menos de un minuto, que contará segundo a segundo. Así mismo, deberá obligar al usuario a introducir un número como input.