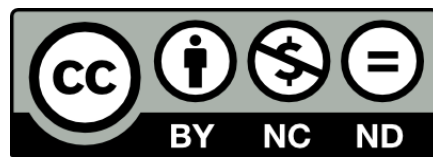


PRÁCTICAS DE PROGRAMACIÓN CON PYTHON



Javier Fernández Panadero



versión 09 septiembre 2022

Fuente del logo: <https://www.python.org/community/logos/>

ÍNDICE

ÍNDICE	1
Introducción	3
Herramientas	4
Hola mundo!	5
Primer objetivo cumplido.	6
INPUT. El usuario también tiene cosas que decir	7
Objetivo cumplido:	10
Variables, tipos y convenciones	11
Objetivo cumplido	12
Variemos las variables.	13
Objetivo cumplido	22
CONDICIONALES. ¿Qué pasaría si...?	23
Objetivo cumplido	30
Listas. Hay vida más allá de las variables	31
Objetivo cumplido	35
ORIENTACIÓN A OBJETOS.	36
Objetivo cumplido	38
BUCLES. No les des más vueltas.	39
BUCLES ANIDADADOS	49
BUCLES CONDICIONALES	51
Objetivo cumplido	52
BREAK Y CONTINUE. Interrumpiendo bucles.	53
Objetivo cumplido	60
FUNCIONES. Los conjuros.	61
Objetivo cumplido	66
LIBRERÍAS. Alguien ya ha lavado los platos...	67
Objetivo cumplido	72
MANEJO DE ERRORES. Cuenta con los fallos	73
Objetivo cumplido	78
Diccionarios. Otros tipos de datos	79
Objetivo cumplido	84

Acceso a ficheros	85
Leer un archivo.	88
Uso del comando with	89
Escribir un archivo	91
Escribir un archivo, añadiendo información	91
Leer partes del archivo	92
Objetivo cumplido	94
Despedida:	95
Otros programas de ejemplo	96
Master Mind	96
Cómo usar Python para tratar datos en SPSS	98
Crear exámenes tipo test en Moodle o papel con datos variables	98
Para ampliar	98

Introducción

Python es un lenguaje muy de moda en estos momentos, de propósito general y con una buena curva de aprendizaje, por lo que resulta bastante interesante para iniciarse en la programación con código.

Estas son unas **prácticas sencillas para personas sin nociones previas ni de programación ni del lenguaje Python.**

Muy útiles para prácticas con alumnos, menos extensas que otros manuales pero que **cubren bastantes temas en un tiempo reducido.**

No es un manual exhaustivo, pero da una buena primera visión.

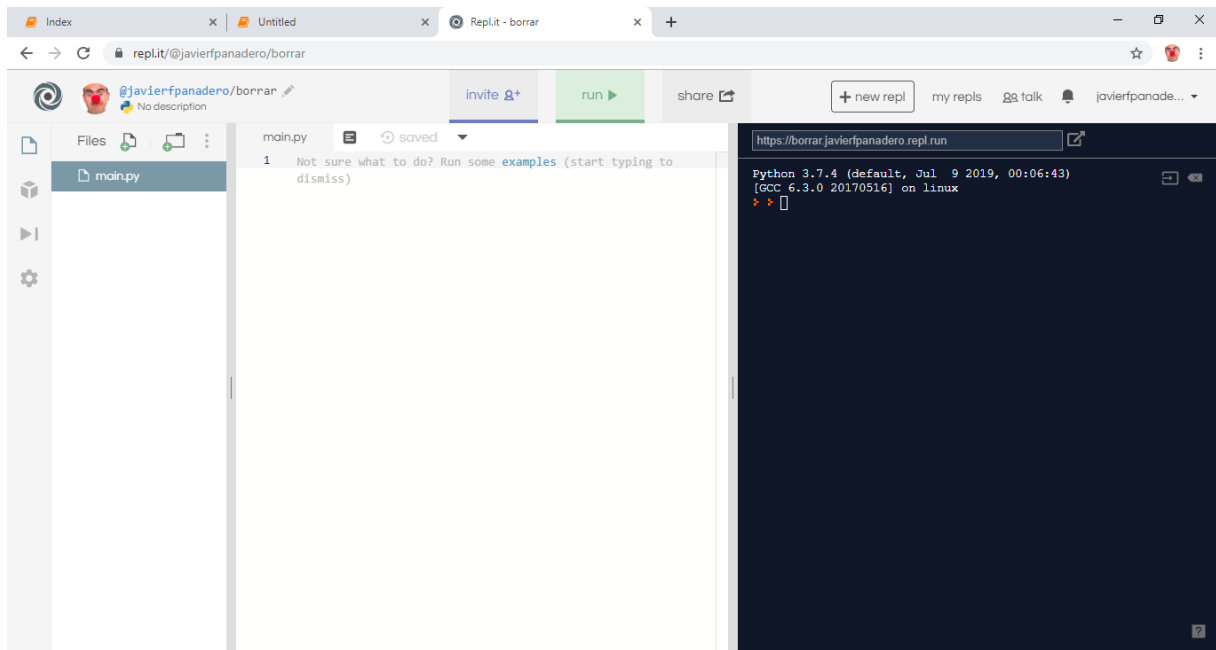
Si os han sido útiles se agradece la difusión y **si queréis podéis echar un cable** con un “café” por aquí <https://ko-fi.com/javierfpanadero>



Herramientas

Por comodidad y facilidad de acceso, vamos a usar un entorno de programación online. En nuestro caso, [Repl](#), pero también es muy interesante [Jupyter](#), donde podéis crear un “bloc de notas” con código insertado, o bien podéis instalar [Python](#) directamente en vuestro ordenador y trabajar en local.

Al crear vuestra cuenta en repl y pulsar arriba NEW REPL aparece esta pantalla



Como podéis ver en la imagen tenemos tres zonas diferenciadas, a la izquierda los archivos que compondrán nuestro proyecto (uno, de momento), en el centro el editor donde escribiremos los programas y a la derecha la ventana del intérprete de Python donde veremos la salida de nuestro programa. En el intérprete también podremos introducir datos durante la ejecución o escribir comandos sueltos directamente.

Los programas escritos podéis guardarlos como un archivo con extensión .py para luego abrirlos en otro entorno y ejecutarlos allí.

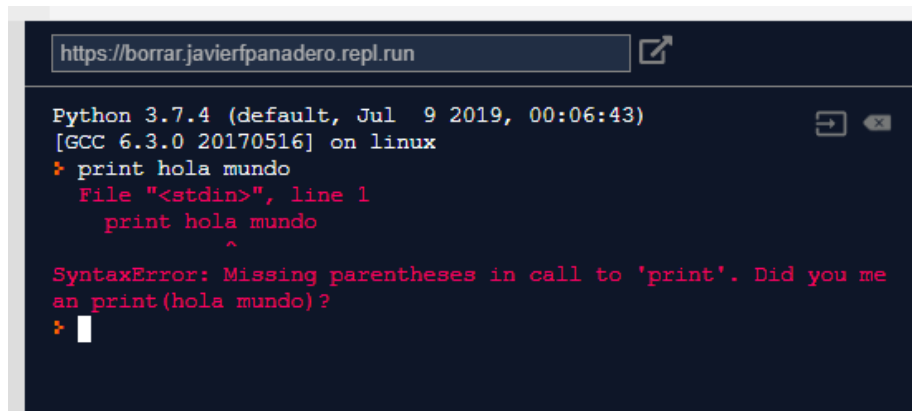
Es importante que mantengamos la edición legible y clara (en Python es crucial la indentación adecuada -el espacio que hay al principio de cada línea-), así que activad el autoformato del editor que uséis, para que os ayude.

Hola mundo!

Es tradición que el primer programa que se escriba cuando uno aprende un lenguaje de programación cualquiera sea aparezca por pantalla este saludo. Así que, ¡a ello!

El comando para poder “escribir” en pantalla es PRINT

Así que será tan sencillo como poner **print** y después lo que queramos que salga por pantalla. Hagámoslo directamente en la ventana “negra”, la del intérprete de Python.

A screenshot of a web-based Python REPL interface. The address bar shows 'https://borrar.javierpanadero.repl.run'. The terminal window title is 'Python 3.7.4 (default, Jul 9 2019, 00:06:43) [GCC 6.3.0 20170516] on linux'. The user has entered the command 'print hola mundo'. The interpreter has responded with a syntax error: 'SyntaxError: Missing parentheses in call to 'print'. Did you mean print(hola mundo)?'. The error message is displayed in red text, and a cursor is visible on the line following the error.

```
https://borrar.javierpanadero.repl.run

Python 3.7.4 (default, Jul 9 2019, 00:06:43)
[GCC 6.3.0 20170516] on linux
> print hola mundo
File "<stdin>", line 1
    print hola mundo
    ^
SyntaxError: Missing parentheses in call to 'print'. Did you mean
print(hola mundo)?
> 
```

¡ENHORABUENA!

Acabáis de recibir **vuestro primer mensaje de error**.

¡Bienvenidos a la programación por código! Veamos qué ha pasad.

En cualquier idioma o lenguaje tienes:

- Lo que quieres decir
- La forma de decirlo

... y puede haber errores en ambas cosas.

Veamos un ejemplo:

Si quiero que me des la sal y te digo “Dame la \$AL” habrá un error ortográfico (sintáctico, se dice en programación) y será una forma incorrecta de pedirla.

Pero si quiero la sal y te digo “Dame el azúcar” también tendremos un problema.

Si ya conocéis la programación en bloques (como Scratch), recordaréis que allí era imposible cometer “errores sintácticos” (SyntaxError) porque no se podían poner los bloques equivocados en los sitios equivocados. Scratch no te deja.

Aunque sí podía ocurrir que el programa funcionara de una manera que no queríamos (traernos el azúcar en lugar de la sal).

El intérprete de Python se quejará si escribimos Python con “faltas de ortografía” y nos regañará, diciéndonos en qué línea de nuestro programa hemos cometido el error y en qué consiste. A veces los mensajes son difíciles de entender para los novatos, pero otras veces ayuda mucho a encontrar el error.

Pero OJO.

El intérprete de Python NO se quejará si el programa que has escrito está bien escrito, aunque no haga lo que tú querías que hiciera. Sólo si lo escribes mal. **Para comprobar que funciona como debe hay que hacer PRUEBAS.**

Pero **NO OS PREOCUPÉIS**, lo de induciros a error ha sido cosa mía para soltaros este rollo.

¡Hagámoslo bien!

Volved a escribir PRINT

Pero poned las palabras que queráis ver escritas en pantalla dentro de un paréntesis y entre comillas (simples o dobles, funcionan ambas)

Algo así como:

```
print('Hola Mundo')
```

Ahora sí, verdad. Mirad debajo del comando que habéis escrito y veréis que os sale la frase que habéis mandado sacar por pantalla.

PRINT es lo que llamamos una **FUNCIÓN**.

Me gusta decir que una función es como un CONJURO. Cuando lo dices, pasa algo. En este caso, que salen cosas por pantalla.

Sabemos que **algo es una función porque va seguido por un paréntesis**, aunque esté vacío.

En la función PRINT debe ponerse dentro del paréntesis lo que quieras que aparezca en pantalla.

Primer objetivo cumplido.

1. Ya sabemos sacar texto por pantalla.
2. Lo que es un SyntaxError.
3. La importancia de la sintaxis para que un programa funcione.
4. La importancia de hacer pruebas para saber que funciona como debe.

INPUT. El usuario también tiene cosas que decir

Ya sabemos hacer que el ordenador “hable” pero, ¿podría el usuario contestarle en tiempo de ejecución? La respuesta es que sí, claro.

Para eso usamos la función INPUT.

Como es una función, ya sabes que debe ir seguida de unos paréntesis, así que escribiremos:

```
input ( )
```

(Ojo, sin espacio entre la t y el paréntesis)

Esta función hará aparecer en pantalla un cursor parpadeante y la posibilidad de que escribamos algo. Emocionante.

Pero claro, ¿dónde va a guardarse lo que yo escriba? Necesitamos reservar un espacio de memoria donde podamos guardarlo y un “indicador” para recordar donde está y poder ir luego a buscarlo, si queremos usar ese dato.

Y así llegamos al concepto de **VARIABLE**.

Digamos que una variable es una CAJA donde voy a almacenar un dato.

Ojo, en una variable no confundir estas tres cosas:

- Tipo de la variable: Si almacena números enteros, decimales, letras...
- Nombre de la variable: Indicador por el que la identifico
- Valor de la variable: el dato que guarda dentro.

Por ejemplo: Puedo tener una variable llamada **edad**, que guarda un dato de tipo **número entero** y que toma el valor 47, en este momento (porque las variables pueden cambiar de valor... claro, son va-ria-bles).

Aunque hablaremos más adelante de funciones, permíteme adelantarte algo.

- Hay conjuros, digo funciones, que se invocan y ya.

Por ejemplo, si digo, ¡Enciéndase la luz! Pues, la luz se enciende y listo.

- Pero hay conjuros que, cuando se invocan, DEVUELVEN ALGO.

Por ejemplo, si digo ¡Voy a cagar! Pues... un regalo viene de camino.

A este tipo de funciones se les llama FUNCIONES CON RETORNO, y más te vale esperar con un orinal, digo con una variable, para que lo que devuelven caiga dentro.

- De esta forma, PRINT, que es una función sin retorno se invoca así:

```
print('Hola Mundo')
```

- En cambio, INPUT, que es una función con retorno, se invoca de forma que lo que devuelva caiga ya dentro de una variable

```
orinal=input()
```

Y así me aseguro que lo que escriba el usuario acaba dentro del... orinal.

¡Procedamos!

Escribamos un programa que pregunte el nombre al usuario y le conteste hola y su nombre. P.ej: Hola, María.

Está uno tentado de empezar con un INPUT, claro, algo así como

```
nombre = input()
```

Pero escribidlo en la ventana blanca y ejecutadlo pulsando en RUN. Efectivamente, aparece el cursor para escribir, pero, ¿cómo sabe el usuario lo que el programa espera de él?

Así que lo más “educado” es empezar sacando por pantalla un mensaje para que el usuario sepa de qué va la historia.

```
print('¿Cómo te llamas?')
```

```
nombre = input()
```

Haremos mucho esto cuando queramos recabar información del usuario¹.

ATENCIÓN, PREGUNTA: ¿Os ha respondido “Hola” y vuestro nombre?

¿No?

¿Está funcionando mal? En absoluto.

Hay una cita que repito mucho.

¹ Hay una manera de hacerlo en una sola línea:

```
nombre=input('¿Cómo te llamas?')
```

Lo MEJOR de la programación es que hace lo que dices. Lo PEOR de la programación es que hace lo que dices.

Recordad esta cita... y usadla mucho si sois profesores.

¿Por qué es lo mejor? Porque los programas no se vuelven locos y hacen lo que les parece, independientemente de lo que yo les haya ordenado (saludos, Skynet)

¿Por qué es lo peor? Porque no va a ayudarme adivinando qué es lo que quiero hacer, se lo tengo que decir explícitamente.

Pero esta es la única manera de tener nosotros el control.

Repasemos nuestro programa:

- Le hemos dicho que imprima una pregunta
- Le hemos dicho que recoja en una variable el nombre que escribamos.

Pues eso ha hecho, ni más ni menos. No ha dado una salida por pantalla final, porque NO SE LO HEMOS DICHO, pero tiene almacenado nuestro nombre en la variable nombre.

¿Queréis comprobarlo?

En la ventana de comandos teclead **nombre** y pulsad intro.

Aparece el valor de la variable nombre. En mi caso: 'Javi'.

Queremos que saque un mensaje en pantalla, pues PRINT, ya sabemos... pero ojo.

En el mensaje hay dos cosas muy distintas, una es la palabra "Hola" que será siempre así, pero quiero que después añada el valor de una variable.

Si escribo

```
print('Hola nombre')
```

Veré con tristeza que aparecerá en la pantalla: **Hola nombre**

Claro, saca por pantalla lo que esté entre las comillas. Así que **nombre** debe estar fuera de las comillas y, si usamos la sintaxis de Python correcta, separada por una coma de la cadena 'Hola'.

Quedará así

```
print('Hola', nombre)
```

Y si quisierais añadir texto después de esta variable, pues de nuevo comillas.

```
print('Hola', nombre, 'pareces más bonito cada día')
```

Objetivo cumplido:

Sabemos:

- Recoger información del usuario en tiempo de ejecución.
- Lo que es una función con retorno y sin retorno.
- Lo que es una variable.
- Distinguir: tipo, valor y nombre de una variable.
- Recoger en una variable el resultado de una función con retorno.
- Dar salida por pantalla combinando “palabras” y valores de variables.
- Que el programa hará lo que le pido, ni más ni menos y eso me da el control.

Variables, tipos y convenciones

Python es un lenguaje de “tipado dinámico”.

Recuerda nuestro programa de antes, nosotros metimos una “palabra” en la variable **nombre** “sin avisar”. En otros lenguajes hay que “crear” las variables y especificar qué tipo de datos van a contener (números, palabras..). Python se va adaptando a los valores que le vamos metiendo y así asume que serán de un tipo u otro según el tipo del valor dado. Esto es muy cómodo para el programador novel (puede acarrear problemas en programas complejos).

Hablemos un poco en detalle de los tipos (y así podré dejar de poner “palabras” entre comillas...).

Uno de los tipos de variables que más usaremos son los enteros (integer en inglés, abreviado como **int**). Ya sabéis, números sin decimales, positivos y negativos.

Si necesitamos números decimales, usaremos “punto flotante” (**float**).

Tenemos las cadenas de caracteres (en inglés **string**), conjuntos de símbolos que incluyen las letras mayúsculas y minúsculas, los signos de puntuación, los caracteres especiales, etc. Estas serán nuestras “palabras” y frases.

También se puede tener una variable en la que se vaya a guardar sólo un carácter (**char**)

Las hay que almacenan nada más que un bit, un uno o un cero (**boolean**) y algunos tipos más...

En nuestro ejemplo, la variable nombre era de tipo string.

Ya has visto que llamamos a la variable como mejor nos pareció, pero hay ciertas reglas.

- No podemos usar palabras “reservadas” del propio lenguaje de programación (print, input... no son nombres de variables permitidos)
- No debe incluir espacios (**edad del usuario** no es un nombre válido)
- Sólo pueden contener letras, números y el guión bajo
- No pueden empezar con un número

Aunque se pueden usar mayúsculas, y Python es muy cuidadoso con eso, entendiendo que *Nombre*, *NOMBRE* y *nombre* son variables DISTINTAS, lo mejor que podemos hacer es no liarnos y seguir las recomendaciones.

Por ejemplo, es típico empezar con minúsculas, como hicimos en **nombre**.

Y usar alguno de estos dos estilos:

numero_matricula_usuario

numeroMatriculaUsuario

Como ves no he usado tildes, muy adrede, es mejor usar sólo caracteres internacionales (evitando las ñ, por ejemplo) que podrían dar errores en algunos sistemas. Para la ñ hay quien, para no escribir **año**, usa **anio**, o bien, **anyo**. Vaya, pero también podéis poner **year**.

En programas elaborados es muy conveniente elegir nombres de variables que nos den una pista de qué están representando (**contador**, **edad**, **nombre**) en lugar de usar cosas como **x**, **azz32**, **c3**.

Objetivo cumplido

- Hay diversos tipos de variables (int, float, strings, etc.)
- No hace falta especificarlo en Python, toman el tipo del valor (tipado dinámico)
- Cómo elegir correctamente el nombre de las variables

Ampliación/Disclaimer

Por la naturaleza inicial de este manual y para que pudiera servir de base para el estudio de otros lenguajes, hemos mantenido la metáfora de la “caja” para las variables.

Pero si ya sabes algo de programación, en Python hay unas particularidades que no comparte con otros lenguajes como C o Java y que resultan muy importantes a la hora de manejar referencias, objetos mutables o inmutables.

En este [brillante vídeo](#), [Ned Batchelder](#) te lo explica a la perfección.

Variemos las variables.

Ya sabemos pedir al usuario datos, e incluso podemos devolvérselos combinados con cadenas de caracteres... pero usamos los datos tal y como nos los dieron.

Vamos a escribir un programa que pida al usuario su nombre y año de nacimiento para decirle al final: Hola, Paco, tienes 45 años.

Para pedirle el nombre y el año de nacimiento usamos la misma estructura que antes. Una línea con un PRINT para que sepan lo que queremos y otra línea con un INPUT para recoger lo que nos dicen.

```
print('¿Cómo te llamas?')
nombre = input()
print('¿En qué año naciste?')
year = input()
```

Estupendo. Ahora cómo calculamos la edad.

Bueno, pues sabemos que la edad será el año actual menos el año de nacimiento.

Pero fíjate, esa resta nos da un valor, que es justo lo que queremos, pero que... TENDREMOS QUE ALMACENAR en algún sitio.

Necesitamos una variable más.

Así que la línea siguiente del programa sería:

```
edad = 2022 - year
```

Con esto creamos la variable edad y le ASIGNAMOS el valor de la resta entre el año 2022 y el valor que tenga la variable **year**, que nos acaba de dar el usuario, y que SERÁ DISTINTO EN CADA EJECUCIÓN.

Vaya, vaya... casi está. ¡Saquémoslo por pantalla!

```
print('Hola', nombre, 'tienes', edad, 'años')
```

Pero.... ERROR!!

Tranquilos. LOS ERRORES SON NORMALES. Leamos el mensaje de error.

Traceback (most recent call last):

File "main.py", line 8, in <module>

edad=2022-year

TypeError: unsupported operand type(s) for -: 'int' and 'str'

La línea 8 es donde yo tengo escrita la operación `edad = 2022 - year`

En la última línea me dice que el operador “menos” no puede trabajar con un tipo entero y un tipo string a la vez. ¿CAPASAO?

De nuevo TODA LA CULPA ES MÍA.

El asunto es que **TODAS LAS ENTRADAS POR TECLADO** se toman como **CADENAS DE CARACTERES**.

Así que el año que introdujo el usuario, para Python, son LETRAS (caracteres), y no sabe cómo restar letras y números.

Así que hay que convertir las letras en números.

¿Hay una manera en Python de tomarse 45 como si fuera un número o como si fuera una cadena de caracteres? Sí.

Hay que usar la función **int()**²

Convierte una cadena de caracteres en un número entero (si es posible).

Veamos cómo.

Ve al intérprete (a la pantalla negra)

Escribe `variable1 = '57'` (entre comillas)

Todo ok.

Ahora teclea `variable1`

Verás que aparece `'57'`

² A estos cambios de tipos se les llama CASTING. También pueden hacerse a la inversa, pasando un entero a cadena de caracteres: `str(12)='12'`

Eso es una cadena de caracteres.

Prueba a escribir

2+2

Sale 4. Todo ok.

Pon ahora 2+variable1

Error.

Creemos una variable2 que sea ese número, pero como número, no como caracteres.

Teclea variable2 = int(variable1)

Hecho. Todo bien. Veamos su contenido.

Teclea variable2

57

¡Ojo, sin comillas!

Ahora pon 2+variable2

Ahora sí funciona, sale 59.

Por lo tanto, cuando capturemos un dato del teclado y queramos que sea tratado como un número en el programa, escribiremos.

```
year = int(input())
```

Lo que significa: toma lo que pongan en el teclado, conviértelo en un número entero y almacénalo como tal en la variable **year**³.

Finalmente, si queremos una salida por pantalla, podríamos poner:

```
print('Hola', nombre, 'tienes', edad, 'años')
```

³ Ojo, si intentas convertir a número algo que no puede serlo, como int(Paco), aparecerá un error.

Hagámonos una pregunta, ¿después de unas semanas te acordarás qué significaba esta línea?

```
year = int(input())
```

Juraría que no, pero vaya, otra pregunta. Si lees esta línea en un programa sin saber lo que sabes, ¿serías capaz de interpretarla correctamente?

Sería estupendo poder incluir algo en los programas que nos explicara qué están haciendo pero, si escribimos texto castellano, Python nos dará un error de sintaxis.

Para eso inventamos los **COMENTARIOS**.

El símbolo # nos permite añadir comentarios al programa y que Python sepa que debe ignorar lo que va después de la almohadilla.

Por ejemplo, en nuestro programa pondríamos.

```
year = int(input())  
#int convierte en entero la entrada de teclado que de otro modo  
sería un string
```

Al leer los comentarios, entendemos el programa, si no es nuestro, o recordamos qué habíamos hecho, si somos los autores. Da un poco de pereza hacerlo, pero son extremadamente útiles.

Con esto en mente, y comentando a lo bestia, el programa de antes quedaría así:

```
'''Pregunta nombre y año de nacimiento y calcula la edad'''  
  
print('¿Cómo te llamas?') #saca por pantalla esta pregunta  
nombre = input()         #lo que se escriba en el teclado  
se mete en la variable nombre  
print('¿En qué año naciste?') #saca por pantalla esta pregunta  
year = int(input())       #int convierte en entero la entrada de  
teclado que de otro modo sería un string  
  
edad = 2022 - year        #calculamos la edad restando del  
año actual
```

```
print('Hola', nombre, 'tienes', edad, 'años') #sacamos por
pantalla los resultados
```

Quizá te suene muy lioso, sobre todo porque estamos explicando cosas que ya son bien conocidas. Igual podíamos dejarlo en:

```
'''Pregunta nombre y año de nacimiento y calcula la edad
Javier Fernández Panadero @javierfpanadero'''

print('¿Cómo te llamas?')
nombre = input()
#nombre usuario
print('¿En qué año naciste?')
year = int(input())
#int convierte en entero la entrada de teclado que de otro modo
sería un string

edad = 2022 - year
#edad usuario (en el 2022)
print('Hola', nombre, 'tienes', edad, 'años')
```

Más sencillo y con información suficiente para entenderlo.

Has podido ver que empezamos el programa explicando qué hace. Aquí también se puede añadir información del autor, fecha, contacto y otros detalles que se consideren relevantes.

Has visto también que cuando vamos a poner un comentario que ocupa más de una línea usamos tres comillas sencillas.

Ya hemos visto que podemos restar una variable numérica de otro número, también las podemos sumar, multiplicar o dividir, pruébalo si quieres en el intérprete.

Hay otras operaciones muy comunes en programación que merecen mencionarse.

Módulo, resto, residuo...

Es el resto de una división

Si dividimos 7 entre tres, dará como cociente 2 y como resto 1

Escribe en la consola `7 % 3` pulsa intro y verás que da como resultado 1

División entera

Nos da el cociente de la división sin obtener decimales.

Si escribes en la consola **7 // 3** obtendrás 2

En cambio si escribes **7 / 3** obtendrás 2.333333333

Potencia

Si escribes en la consola **2**3** obtendrás 8 (dos elevado a tres)

Un solo asterisco te dará la multiplicación **2*3** y saldrá 6

Para las operaciones hay un orden de prevalencia, como en las matemáticas comunes, por lo cual debes tener cuidado y usar paréntesis, así como **asegurarte de que lo que escribes funciona como deseas (¡Haz tests!)**.

Las operaciones con números te son bien conocidas... pero también pueden hacerse **OPERACIONES CON CADENAS DE CARACTERES**.

Por ejemplo, ¿qué pasará si escribes 'Mamá' + 'Papá'?

Prueba...

Efectivamente, **CONCATENA** las dos cadenas, haciendo una cadena mayor.

Intenta hacer 'Mamá' * 5 a ver qué pasa...

Hay más operaciones, pero sigamos adelante.

Sigamos hablando de variables. Recuerda que dijimos que **una variable era como una CAJA que almacenaba un valor**. Ese valor se **ASIGNA** usando el símbolo igual (=)

Por lo tanto, **x = 5** se leería

“La variable x tomará el valor 5”

Déjame plantearte una pregunta, ¿qué pasa después de estas líneas de código?

```
x = 5
```

```
x = 3
```

¿Cuánto vale x? ¿Sigue con el valor 5, el primero que recibe? ¿Toma el último valor asignado? ¿Los va acumulando y valdrá entonces 8? ¿Da un error al tratar de asignar un valor a una variable que ya tiene un valor?

La respuesta correcta es que **las variables se SOBREENSCRIBEN**, así que toman el último valor que se les asigne. En nuestro caso, x valdría 3.

Dime ahora, ¿qué pasa después de estas líneas de código?

```
x = 3
x = x + 1
```

Te confieso que cuando empecé con la programación la última línea me hacía estallar la cabeza:

¿Cómo va a ser posible que x “sea igual a” x+1? ¡!!!Es IMPOSIBLE!!!!

Pero ya os he dicho que **EL IGUAL EN PROGRAMACIÓN ES UNA ASIGNACIÓN**.

x = x + 1 se lee así:

“La variable x tomará el valor que tiene ahora más una unidad”

Así que después de las dos líneas de código que escribimos, x valdrá 4.

Estas expresiones son muy comunes y debes familiarizarte con ellas⁴.

Recordemos una frase que decíamos antes: **Cada vez que tengamos que recordar un valor necesitamos una variable**.

Te propongo hacer un programa que intercambie el valor de dos variables.

⁴ Tan frecuentes son que se abrevian así

x+ = 1 significa x = x + 1

OJO que no es lo mismo +x=3 que x+=3... os dejo a los curiosos averiguar qué pasa ;)

El primer intento que haría un principiante (ojo, está mal) es posible que fuera este:

```
a = 3
b = 4
print('El valor de a es ', a)
print('El valor de b es ', b)
a=b
b=a
print('El valor de a es ', a)
print('El valor de b es ', b)
```

Pero esto es lo que obtengo en la consola

(Antes del cambio)

```
El valor de a es  3
El valor de b es  4
```

(Después del cambio)

```
El valor de a es  4
El valor de b es  4
```

De nuevo, un ejemplo de **programa con la sintaxis correcta, pero que no actúa como debe**. Recuerda, un programa sin errores sintácticos no es un programa que haya probado que funciona adecuadamente.

Veamos qué ha pasado, instrucción por instrucción y viendo qué valores están tomando las variables después de cada paso.

Instrucción	Valor de a	Valor de b	Comentarios
a = 3	3	No existe aún	Asignación de a
b = 4	3	4	Asignación de b
a = b	4	4	a toma el valor que tenía b b no cambia su valor
b = a	4	4	b toma el valor que tenía a en el paso anterior, o sea, 4 a no cambia su valor

Como puedes ver, el valor primitivo de `a` se pierde, porque no podemos hacer el intercambio “a la vez”, sólo instrucción por instrucción. Por eso necesitamos guardar el valor primitivo de `a`, antes de sobreescribirlo, para poder luego pasárselo a `b`.

Y, cuando necesitamos guardar algo... necesitamos una variable.

La llamaremos **temp**, nombre típico de variables temporales que no queremos usar más que para cosas intermedias. También es frecuente usar **aux** (auxiliar).

Veamos si funciona.

```
a = 3
b = 4

print('Antes del cambio')
print('El valor de a es ', a)
print('El valor de b es ', b)

temp = a    #guardamos el valor de a en la variable temp
a = b       #asignamos a la variable a al valor que tiene b
b = temp    #asignamos a b el valor que tenía a que estaba guardado en
la variable temp

print('Después del cambio')
print('El valor de a es ', a)
print('El valor de b es ', b)
```

Si lo ejecutáis, en la consola aparece

```
Antes del cambio
El valor de a es 3
El valor de b es 4
Después del cambio
El valor de a es 4
El valor de b es 3
```

¡Ahora sí funciona!

Si analizamos de nuevo el valor de las variables en cada paso lo verás más claro

Instrucción	Valor de a	Valor de b	Valor de temp	Comentarios
a=3	3	No existe aún	No existe aún	
b=4	3	4	No existe aún	
temp=a	3	4	3	Guardamos valor de a en temp
a=b	4	4	3	Sobreescribimos a
b=temp	4	3	3	Recuperamos el valor que tenía a para asignárselo a b, gracias a que estaba guardado en temp

Seguiremos jugando con las variables más adelante, pero ahora, avancemos.

Objetivo cumplido

- Cómo asignar un valor a una variable como resultado de un cálculo
- Saber que las entradas de teclado se toman como cadenas de caracteres (strings)
- Cómo pasar una string a un entero, función int()
- Cómo ayudarnos a entender un programa usando los comentarios
- Podemos hacer operaciones con otros tipos de variables, como con las strings
- Saber que $x = x+1$ NO es una igualdad, sino una asignación de valor a x al valor que tenía antes más una unidad.
- Cómo evitar sobreescribir variables que podamos necesitar luego (usando variables auxiliares)

Nota: Este ejemplo era sólo para que vieras cómo funciona la asignación y la sobreescritura. El día que quieras intercambiar el valor de dos variables solo tienes que hacer $a, b = b, a$
¡Pruébalo!

CONDICIONALES. ¿Qué pasaría si...?

Nuestros programas, hasta ahora, hacen siempre lo mismo, sean cuales sean los datos que les introduzcamos, pero es muy útil que puedan tomar decisiones diferentes según sean los datos que reciben o los resultados que van obteniendo.

Por ejemplo, recordemos este programa nuestro que nos preguntaba el año de nacimiento y nos decía *Hola Javi tienes 47 años*, podría ser divertido que nos diese diferentes respuestas según los años que nos calcule (por ejemplo si somos mayores de edad o no). Para eso necesitamos los CONDICIONALES.

Un condicional tiene tres “partes”.

La condición que queremos comprobar (en este caso $edad > 18$)

Las acciones que tomamos si se cumple la condición (nos dirá que somos viejos)

Las acciones que tomamos si NO se cumple la condición (dirá que somos jóvenes)

Traigamos y retoquemos el antiguo programa

```
'''Pregunta nombre y año de nacimiento y calcula la edad
Javier Fernández Panadero @javierfpanadero'''

print('¿Cómo te llamas?')
nombre = input()                #nombre usuario
print('¿En qué año naciste?')
year = int(input())             #int convierte en entero la entrada de
teclado que de otro modo sería un string

edad = 2019 - year              #edad usuario (en el 2019)
print('Hola', nombre, 'tienes', edad, 'años')
```

Nos sirve todo, menos la línea final, ahí es donde vamos a meter el condicional y daremos otras salidas por pantalla diferentes a la que tenemos aquí.

No la borreís, os voy a enseñar **un truco** muy común cuando estamos probando un programa y queremos “activar” y “desactivar” líneas de código, para ver qué pasa cuando las quitamos.

¿Recordáis que, cuando poníamos la almohadilla(#), Python ignoraba lo que se escribía después y que nos servía para poner comentarios? Pues podemos poner una almohadilla delante de la línea que queremos anular temporalmente y Python la ignorará. A esto se le llama **COMENTAR CÓDIGO**.

```
'''Pregunta nombre y año de nacimiento y calcula la edad
Javier Fernández Panadero @javierfpanadero'''

print('¿Cómo te llamas?')
nombre = input()           #nombre usuario
print('¿En qué año naciste?')
year = int(input())        #int convierte en entero la entrada de
teclado que de otro modo sería un string

edad= 2019 - year          #edad usuario (en el 2019)
#print('Hola', nombre, 'tienes', edad, 'años')
```

Así es como si no existiera, pero la tenemos por ahí por si queremos volver a activarla o tomarla como referencia. Por supuesto, cuando terminemos el programa podemos eliminar estas líneas por si pudieran confundir a un futuro lector del código.

Bien, vayamos al condicional

Debemos empezar con la palabra reservada **if**

Después ponemos la condición

Después añadimos dos puntos (:)

Haced esto y dadle a intro.

¿Veis que al cambiar de línea el cursor ha avanzado un poco? Se llama **INDENTACIÓN**⁵ y sirve para tener claro que las líneas siguientes tienen que ver con el **if**, que es un pequeño bloque de código dentro de ese condicional.

En esa línea siguiente ponemos las instrucciones que queremos se ejecuten cuando se cumpla la condición (puede ser una o muchas)

⁵ Suelen (y deben) usarse cuatro espacios.

```
if edad > 18:  
    print('Hola', nombre, 'eres muy viejo')
```

Prueba el código, recuerda que la ausencia de errores no asegura que funcione bien. Mete dos años de nacimiento diferentes, uno que nos dé mayor de edad y otro que no. **Cubre todos los casos cuando hagas pruebas.**

Como ves, en el caso de que se cumpla la condición, ejecuta el código “interior” del condicional y nos imprime este “simpático” mensaje.

Pero, en el caso de que NO se cumpla. No hace nada. Esto no es lo que queremos. Para remediarlo, tenemos **else (en caso contrario...)**.

Añadiríamos al programa que tenemos las siguientes líneas.

```
if edad>18:  
    print('Hola', nombre, 'eres muy viejo')  
else:  
    print('Hola', nombre, 'eres muy joven')
```

Date cuenta de que

- **else** está a la altura del **if** correspondiente
- después de **else** tenemos dos puntos porque también viene un código “interior”
- el código “interior” de **else** también está indentado.

Lo que escribimos justo después del **if** debe ser una EXPRESIÓN LÓGICA, algo que sea como una pregunta que pueda responderse con SÍ o NO, VERDADERO o FALSO. En nuestro ejemplo ¿Es **edad** mayor que 18? Efectivamente, puede responderse VERDADERO o FALSO.

También valdrían expresiones complejas. Por ejemplo, si queremos dejar pasar a un chaval a una atracción un poco peligrosa, quizá queramos que sea mayor que 10 años o que su altura sea mayor que un metro. Podríamos expresarlo así:

edad > 10 **or** altura > 1.0

Por ejemplo:

```
altura = 1.2
edad = 10

if edad > 9 or altura > 1.0:
    print('pasa')
else:
    print('no pasa')
```

Con la altura y la edad que he puesto, el programa me dice 'pasa', porque tiene más altura y más edad de la exigida.

Así que, funciona bien con un ejemplo y no da errores sintácticos, ¿puedo quedarme tranquilo? **¿Cuántos casos diferentes hay?**

A mí me salen cuatro (refiriéndome a edad y altura):
menos-menos, menos-más, más-menos y más-más

Cambiad a mano los valores para ejemplificar los cuatro casos y ver que se comporta adecuadamente. **Y esto, queridos míos, sí que es comprobar que un programa funciona debidamente.**

De la misma forma que hemos usado "or" podemos usar "and" o "not" y crear expresiones todo lo complejas que necesitemos.

Pero quería pararme un momento para resaltar **un caso que suele dar problemas**:
¿cómo escribir que quiero comprobar si la edad es igual a 18?

Lo primero que se nos ocurre es poner *if edad = 18*: pero esto está MAL.

Recordad que el IGUAL lo usábamos para ASIGNAR.

Para COMPARAR usaremos un DOBLE IGUAL⁶

Lo correcto es, por ejemplo:

```
print('Escribe tu nombre')
nombre = input()
if nombre == 'Javi':
    print('Te llamas igual que yo!')
```

⁶ También son símbolos válidos para expresiones: mayor o igual (\geq), menor o igual (\leq) y distinto (\neq).

Ya conoces todo lo que hay en este programa, pero permíteme señalar unos detalles.

- Las cadenas de caracteres se escriben entre comillas
- PRINT no mira si lo que hay entre las comillas es código Python o español, o tiene faltas de ortografía como es el caso (falta el signo j), simplemente pone los caracteres que haya.

Volvámonos locos... pongamos un condicional dentro de otro, así tendremos lo que llamamos, **CONDICIONALES ANIDADOS**.

Por ejemplo: Quiero celebrar mi cumpleaños y, por supuesto, me gustaría que vinieran más de cinco personas, pero no tengo sitio para más de diez.

```
'''El número de invitados no puede ser superior a 10 ni inferior a 5'''
```

```
print('¿Cuántos invitados vienen')
invitados=int(input())

if invitados > 5:
    if invitados < 10:
        print('Hay fiesta') #caso entre 5 y 10
    else:
        print('No hay fiesta') #caso mayor que 10
else:
    print('No hay fiesta') #caso menor que 5
```

Comprueba el funcionamiento con un valor de cada zona interesante (menos que cinco, entre cinco y diez, más de diez).

Analicemos el código.

Primer **if**, ¿estamos por encima de 5?, si es así ejecutamos el código interior, si no fuera así, nos iríamos al **else** que está a su altura (el último).

Si entramos en el primer **if** es que estamos por encima de 5, pero ahora nos preguntamos: sabiendo que estamos por encima de 5, ¿estamos por debajo de 10? Si fuera verdad, estamos en la franja adecuada entre cinco y diez, y si no se cumple que seamos menor que diez, pasaríamos al **else** que está a la altura del segundo **if**.

Igual lo ves más claro, si entendemos **por qué no funcionaría bien el código si no estuvieran anidados**. Pongamos los dos if a la misma altura.

```
'''El número de invitados no puede ser superior a 10 ni inferior a 5'''
print('¿Cuántos invitados vienen')
invitados = int(input())
if invitados > 5:
    print('Caso1')
else:
    print('Caso2')
if invitados < 10:
    print('Caso3')
else:
    print('Caso4')
```

¿Dónde pongo, “hay fiesta” y “no hay fiesta”? Da igual, no va a funcionar en ningún caso. Te pongo un ejemplo para que veas que “casca”. Prueba tú con otros, si crees que van a funcionar, pero recuerda probar los tres casos posibles.

Si los pongo así, por ejemplo:

```
'''El número de invitados no puede ser superior a 10 ni inferior a 5'''
print('¿Cuántos invitados vienen')
invitados = input()
if invitados > 5:
    print('Hay fiesta')
else:
    print('No hay fiesta')
if invitados < 10:
    print('Hay fiesta')
else:
    print('No hay fiesta')
```

Imagina que tengo 3 invitados.

El primer condicional me dirá que no hay fiesta, pero el segundo condicional me dice que sí, porque tres invitados es menor que diez. Al NO estar anidados no estoy mirando si es menor que diez SABIENDO ya que es mayor que cinco.

A veces abreviamos estos condicionales con el comando **elif** que sería **else...if**

Algo así como

¿Se cumple esta condición? No

Sabiendo que no se cumple la primera condición, ¿se cumple la segunda?

Por ejemplo, adivina un número entre uno, dos y tres.

```
'''Adivina un número entre 1, 2 y 3.  
El número a adivinar es el 2'''  
print('Dime un número entre el uno y el tres')  
test = input()  
if test == '1' or test == '3':  
    print('Has fallado')  
elif test == '2':  
    print('Has acertado')  
else:  
    print('No has escrito un número entre el uno y el tres')
```

Se leería así:

¿Es test igual a uno o a tres? Si es así, di que “ha fallado”.

(elif) SABIENDO que no es igual a uno o a tres, ¿es igual a dos? Si es así, di que “ha acertado”.

(else) En CUALQUIER OTRO CASO, di que no ha escrito bien lo que le pedíamos.

En este caso es sencillo porque las opciones son excluyentes, pero en otros casos hay que ser cuidadoso para escoger adecuadamente las condiciones y en el orden que se piden, quién se anida en quién...y, finalmente, **hacer pruebas para todos los casos posibles**.

Esto podría usarse para crear un “menú”.

Hagamos un programa que te pida dos números y te dé la posibilidad de elegir sumarlos, restarlos o multiplicarlos.

```
'''Pide dos números, y te da a elegir sumarlos, restarlos o
multiplicarlos'''
print('Dime el primer número')
a = int(input())
print('Dime el segundo número')
b = int(input())
print('Elige suma(s), resta(r) o multiplicación(m)')
operacion = input()
if operacion == 's':
    print('El resultado es', a + b)
elif operacion == 'r':
    print('El resultado es', a - b)
elif operacion == 'm':
    print('El resultado es', a * b)
else:
    print('No has escogido correctamente la opción')
```

Aunque ya todo es conocido, permíteme que te recuerde que:

- Para poder usar como números enteros los números que teclea el usuario hay que pasarlos por la función **int** porque si no los guardaría como string.
- Cuando miro qué ha escogido el usuario, lo comparo con 's', 'r' o 'm' porque son strings
- Es bueno poner una opción **else** final porque así, siempre ocurrirá algo. Nos aseguramos que el programa dará alguna respuesta.

Objetivo cumplido

- Comentar código para “desactivarlo” temporalmente durante pruebas
- Cómo evaluar una condición y hacer unas cosas si se cumple y otras si no.
- La importancia de la indentación para ver los códigos “interiores” a estructuras de control, como los condicionales y que van precedidos de dos puntos.
- La importancia de las pruebas del código que contemplen todos los casos posibles
- Las expresiones lógicas
- La diferencia entre asignación (=) y comparación (==)
- Los condicionales anidados
- Uso de elif

Listas. Hay vida más allá de las variables

Hasta ahora hablábamos de las variables como la forma de almacenar información y decíamos que siempre que quisiéramos guardar algo íbamos a necesitar una variable, pero eso no es del todo cierto, porque existen otras estructuras de datos.

Imagina que queremos guardar todos los nombres de los estudiantes de una clase.

Bien podríamos tener treinta variables, una para cada nombre, pero es cierto que esos nombres tienen cierta “relación” entre sí, no son variables separadas “conceptualmente” como pueda ser mi altura, el PIB de tu país y tu edad. Nadie va a intentar ordenar de mayor a menor esos tres números, por ejemplo, no tiene sentido. En cambio, sí podríamos plantearnos ordenar alfabéticamente los nombres de una lista de una clase. Esos datos tienen cierta “relación” entre ellos.

Bueno pues para eso usamos las listas.

Así que una **LISTA** es un conjunto de datos. Una de sus características es que están **ORDENADOS**. Hay un primer elemento de la lista, un segundo elemento, etc.

Los elementos de una lista pueden ser enteros, caracteres... todos los tipos de los que hemos ido hablando. Incluso, en una misma lista, cada elemento puede ser de un tipo diferente.

La lista recibe un nombre, igual que las variables, y los elementos serán su contenido. Por ejemplo:

```
lista_extraña = [2, 'Hola', 3.5]
```

Teclea esto en **la consola** (ventana negra), para que se quede en la memoria.

Fíjate que usamos los corchetes [] y que los elementos van separados por comas. Es importante también que recuerdes que para los decimales usamos un punto.

Hemos dicho que la lista está ordenada, eso quiere decir que el primer elemento de la lista será un 2, el segundo la string 'Hola' y el tercero el float 3.5... PEEEEERO, se numeran empezando por el cero. Así que, si escribes en la consola cada elemento (identificado por su **índice**) y le das a intro pasará lo siguiente.

Escribe en la consola `lista_extraña[0]` y pulsa intro
Aparecerá 2

Haz lo mismo con `lista_extraña[1]` y `lista_extraña[2]`, verás como obtienes los otros dos elementos.

¿Sería muy absurdo hacer una lista de listas?

Me refiero a una lista en la que cada elemento es, a su vez, otra lista.

No es tan descabellado, puede ser un plano de la clase, con la posición de las mesas y el nombre de los chavales. El primer índice son las filas y el segundo las columnas, por ejemplo.

```
mi_aula = [['Luis', 'María', 'Juan'], ['Gema', 'Celeste', 'Raquel']]
```

(amplía el ancho de la consola si te da problemas al escribir)

Es una clase con dos filas y con tres columnas.

Escríbela en la consola para guardarla en la memoria.

Ahora preguntémonos, ¿quién está en la segunda fila, en la tercera columna?

Pues, COMO SE EMPIEZA DESDE EL CERO, será... `mi_aula[1][2]`

Si lo tecleas en la consola y pulsas intro, verás que aparecerá el nombre correcto.

También te puedes preguntar por todas las personas de la segunda fila, entonces deberás teclear: `mi_aula[1]`

Hablemos ahora de una función muy útil con las listas. Ya hemos visto funciones antes, como **`print()`**, **`input()`**, **`int()`**. Todas seguidas de paréntesis, ¿recuerdas? Porque así sabemos que son funciones.

La función que quería explicaros es **`len()`** y nos da la longitud de una lista, el número de elementos que tiene.

Prueba a poner `len(mi_aula)` y `len(mi_aula[1])`

¿Todo bien?

`mi_aula` es una lista que tiene dos elementos, que son a su vez listas, de acuerdo, pero tiene dos elementos. Por eso `len(mi_aula)` arroja el resultado de 2. Cuando ponemos `len(mi_aula[1])` nos da la longitud del segundo elemento de `mi_aula` que es una lista también, en este caso de tres elementos.

Usaremos mucho el índice que nos marca cada elemento de la lista y la longitud de la lista para movernos por dentro de ellas, pero también hay más funciones que podemos utilizar. Por ejemplo, podemos **CONCATENAR LISTAS**, como concatenábamos cadenas de caracteres, usando el símbolo “+”.

Si escribes esto en la consola `[2,3]+[4,6]` y pulsas intro, te devolverá una lista de cuatro elementos `[2,3,4,6]`

También podemos conseguir **SEPARAR PARTES DE UNA LISTA**.

Por ejemplo: teclea en la consola: `prueba = [3,4,6,8,12]`

Ahora teclea **`prueba[1:3]`** y aparece `[4,6]` ves que el “trozo” empieza en el índice que marca el primer número (1) y se para justo antes del que marca el segundo número, así que sólo llega hasta el elemento con índice 2.

Podemos **SOBREESCRIBIR ELEMENTOS** simplemente asignándoles valor

```
prueba[2] = 'Perro'
```

Teclea prueba ahora a ver qué sale.

También podemos **ELIMINAR** un elemento de la lista **REDUCIENDO SU LONGITUD**

Pon en la consola **`del prueba[2]`** y pulsa intro.

Ahora escribe prueba y verás que el elemento no está y la lista es más corta.

Vamos a hacer un ejemplo de uso de listas. **Vamos a calcular la letra del DNI.**

En España tenemos un número de DNI que acaba en una letra. Hay quien cree que es para “tener más números”, imagina que el DNI fuera 232B, pues habría 232C, 232D, etc. Pero **NO ES ASÍ**.

Esa letra es un carácter de control, se calcula a partir de los números anteriores y sirve para detectar errores.

Imagina que tienes que cargar una multa y alguien teclea mal el número, podría ser que se la pusieran a otro.

Lo que sucede es que cuando uno introduce el número en los programas de gestión de estas cosas, el ordenador calcula la letra y la compara con la que tú has puesto. Si no coincide te dará un mensaje de error.

El algoritmo es sencillo. Se divide el número del DNI entre 23 y se toma el resto (que puede ser desde cero hasta 22). Según sea el resto, se escoge una letra u otra.

RESTO	0	1	2	3	4	5	6	7	8	9	10	11
LETRA	T	R	W	A	G	M	Y	F	P	D	X	B

RESTO	12	13	14	15	16	17	18	19	20	21	22
LETRA	N	J	Z	S	Q	V	H	L	C	K	E

Fuente: [Ministerio del interior.](#)

Recordaréis que teníamos una función que nos daba el resto de una división, se llamaba módulo y se simbolizaba con %.

Pues ya está casi...a ver qué te parece.

```
'''Calcula la letra del DNI usando una lista'''
```

```
letras_dni=['T','R','W','A','G','M','Y','F','P','D','X','B','N',
            'J','Z','S','Q','V','H','L','C','K','E']
```

```
print('Introduce tu número del DNI (sin la letra)')
```

```
num_dni = int(input())
```

```
resto = num_dni%23 #calcula el resto de dividir entre 23
#print(resto)      #si quieres ver el valor del resto descomenta
esta línea
```

```
resultado = letras_dni[resto] #escoge la letra adecuada
```

```
print('La letra de tu DNI es ', resultado)
```

El código no debería ofrecer dudas. Fíjate que te he dejado una línea comentada por si quieres que salga también por pantalla el valor del resto. No es necesario, pero por si quieres comparar en la tabla “a mano” que la letra está bien elegida, ahí está.

Tenemos suerte de que, para nuestra lista, el resto es justo el índice que necesitamos, si fuera uno más o uno menos tampoco sería un problema, habríamos puesto `resultado = letras_dni[resto+1]` por ejemplo.

Este truco de los caracteres de control se usa continuamente en los paquetes de datos que mandamos por la red o en el número de tu cuenta bancaria. Las siglas D.C. significan “dígito de control” y el IBAN también es un código de control.

Objetivo cumplido

- Sabemos que hay otras estructuras de datos aparte de las variables
- Cómo crear listas, sobrescribir elementos o borrarlos
- Cómo referenciar los elementos
- Cómo referenciar sólo una parte de la lista
- Listas de listas
- Calcular la longitud con **len**
- Qué es un carácter de control

ORIENTACIÓN A OBJETOS.

El paradigma de programación orientada a objetos ya tiene unos añitos, intentaré explicar lo básico, para poder funcionar sin que nos liemos mucho.

Permitidme una METÁFORA:

Objeto genérico: Un coche (así, sin más detalle)

Una instancia, un objeto concreto: mi_cacharraco

Atributos, propiedades del objeto: Color, marca, potencia, número de puertas...

Métodos, acciones que puede llevar a cabo mi objeto: arrancar, frenar, acelerar...

¿Para qué sirve esto? Porque alguien YA HA PROGRAMADO EL OBJETO GENÉRICO, EL COCHE! Ha programado lo básico, solo tengo que dar mis detalles.

Sólo tengo que decir que mi coche es un objeto de tipo coche, dar el valor de los atributos y ¡YA PUEDO USAR SUS MÉTODOS!.

Sería algo así como:

Instanciación:

mi_cacharraco es un objeto de tipo coche

Dar valores a atributos:

mi_cacharraco.color=rojo

mi_cacharraco.num_puertas=4

Y ahora si me apetece... invoco a un método (que es una función) y debería ir bien.

mi_cacharraco.arrancar()

RRRROOOM, RRRROOOOMMMMMM

Vale, pues funciona.

Y no he tenido que programar ni como dar color a un coche ni cómo hacer que arranque... de eso ya se ha ocupado quien programó la el objeto genérico coche.

Vayamos a nuestras movidas...

Las listas son objetos.

Tienen atributos como su longitud, el valor de los elementos que la forman...

Y también tienen métodos que podemos llamar, como por ejemplo, agregar un elemento al final de la lista.

Como has visto en el ejemplo del coche, tanto los atributos como los métodos se llaman con el **nombre de TU objeto concreto, un punto, y luego el atributo o el método**.

¿Cómo diferenciar unos de otros? Fácil, los métodos son funciones y, como te dije, van siempre seguidos de paréntesis. ¿Vemos un ejemplo de uso de métodos?

```
'''Ejemplo de uso de un método de las listas'''
nombres=['Paco','María','Raquel']
#al definir nombres como una lista, ya puedo llamar a
cualquiera de los métodos que se aplican sobre listas.
#vamos a añadir un elemento al final de la lista
print('Añadir')
print('Lista previa')
print(nombres)
print('Dame el elemento a añadir')
nuevo = input()
nombres.append(nuevo) #añade elemento al final
print('Lista después de añadir un elemento')
print(nombres)
#fíjate que el método se llama en la lista concreta sobre la
que se quiere usar, NO se pone lista.append, sino
nombres.append
print('Insertar')
print('Lista previa')
print(nombres)
print('Dame el elemento a insertar')
nuevo = input()
print('Dame la posición de inserción')
posicion = int(input())
nombres.insert(posicion, nuevo) #inserta elemento en la
posición dada, ojo a la numeración que empieza por cero
print(nombres)
#de nuevo llamamos al método con el objeto concreto que usamos
```

Si has estado atento, al teclear en el editor “nombre.” te salían un montón de opciones con métodos y ayuda de cómo usarlos.

Hay **infinidad de objetos y métodos** programados para ellos. Según necesites, ve buscando y **mirando las ayudas** para ver cómo usarlos. Por ejemplo, en el método INSERT hay que poner primero el índice y luego el valor, **esto no lo adivinamos, nos lo dice la ayuda**. Acuérdate siempre de que no eres un pionero, eres un principiante. Eso significa que EL PROBLEMA QUE TIENES TÚ es probable que LO HAYA TENIDO MUCHA GENTE ANTES... y que lo hayan SOLUCIONADO.

Así que, ya sabes, cada vez que veas algo de la forma `mi_objeto.metodo()` será una función que se aplica directamente sobre el objeto que hayas creado: “mi_objeto”.

Supongo que ya habrás visto las ventajas de esto. Piensa en el método INSERT, ¿cómo sería hacerlo a mano? Habría que localizar la posición que nos indican, meter ahí el nuevo elemento, y renombrar a los demás para que aumentaran una unidad su índice... parece un poco más difícil que escribir `nombres.insert(2,'Julián')`, ¿no?

Objetivo cumplido

- Qué es la orientación a objetos y por qué es útil (ahorra programar)
- Qué son los atributos y los métodos
- Cómo usar los métodos de un objeto

BUCLES. No les des más vueltas.

La informática nace para tratar automáticamente la información. Nadie en su sano juicio se compraría un ordenador y una impresora para colgar UN cartel en su portal pero, si tengo que escribir trescientas cartas, igual ha llegado el momento de pensar incluso en cómo introducir automáticamente las diferentes direcciones a partir de un fichero. Si la tarea completa me va a llevar cinco minutos no tiene sentido invertir media hora en ver cómo automatizarla, pero si la tarea me llevaría una semana, bien puedo invertir un día en mirar cómo hacerla de forma automática.

Los bucles son estructuras de control que nos permiten repetir tareas que son iguales o con pequeñas variaciones, un número de veces determinado o bien hasta que se cumpla alguna condición.

Por ejemplo, llenar una lista de valores.

Quiero confeccionar una lista con los jugadores de mi equipo de baloncesto.

Podríamos hacerlo paso a paso

```
'''Programa para ver el funcionamiento de distintos bucles'''  
print('Hagamos una lista de jugadores')  
equipo = [] #creamos una lista vacía para poder llenarla  
  
print('Introduce el nombre del primer jugador')  
equipo.append(input())  
  
print('Introduce el nombre del segundo jugador')  
equipo.append(input())
```

Y repetir este grupo de dos líneas tantas veces como jugadores tenga... un tostón.

Funciona mucho mejor así:

```
'''Programa para ver el funcionamiento de distintos bucles'''
print('Hagamos una lista de jugadores')
equipo=[] #creamos una lista vacía para poder llenarla

for i in range(5):
    print('Introduce el nombre un jugador')
    equipo.append(input())

print(equipo)
```

Hablemos del bucle

Primero vemos que aparece una variable que se llama i. Es un contador, para saber cuantas repeticiones llevamos. Las llamamos **iteraciones**.

range(5) me indica que voy a hacer 5 iteraciones en total

Los dos puntos dan paso al bloque de código “interior” que es lo que se va a repetir cinco veces. Sacar por pantalla el mensaje y añadir un elemento a la lista.

Finalmente, y fuera del bucle, escribimos el equipo entero.

Si hubiéramos puesto la última línea dentro del bloque de código del bucle, habría salido en pantalla, en cada iteración, cómo va de lleno el equipo.

Hagámoslo y lo vemos, y de paso hagamos que nos ponga el valor de i en cada iteración y así queda todo claro.

```
'''Programa para ver el funcionamiento de distintos bucles'''
print('Hagamos una lista de jugadores')
equipo=[] #creamos una lista vacía para poder llenarla

for i in range(5):
    print(i)
    print('Introduce el nombre un jugador')
    equipo.append(input())
    print(equipo)
```

Esto me sale por pantalla

Hagamos una lista de jugadores

0

Introduce el nombre un jugador

Javi

['Javi']

1

Introduce el nombre un jugador

Gaby

['Javi', 'Gaby']

2

Introduce el nombre un jugador

Mari

['Javi', 'Gaby', 'Mari']

3

Introduce el nombre un jugador

Pepa

['Javi', 'Gaby', 'Mari', 'Pepa']

4

Introduce el nombre un jugador

Luisa

['Javi', 'Gaby', 'Mari', 'Pepa', 'Luisa']

Así que i empieza en cero y termina en cuatro. A tener en cuenta cuando escribamos los bucles, por ejemplo, cambia el bucle por este (acuérdate de añadir luego imprimir el equipo, claro).

```
for i in range(5):  
    print('Introduce el nombre del jugador', i + 1)  
    equipo.append(input())
```

Verás que LAS ITERACIONES NO TIENEN POR QUÉ SER IDÉNTICAS.

Cada iteración puede variar teniendo en cuenta por dónde va el contador, como en este ejemplo, o por los valores que tomen las variables.

Vamos a hacer **un programa que muestre todos los números pares entre 1 y 100**

```
print('Estos son los números pares entre 1 y 100')
for i in range(1, 101):
    if (i % 2 == 0):
        print (i)
```

Hacemos correr el contador entre 1 y 101 (ya sabes que se para justo antes, por eso para llegar al 100 tenemos que poner 101)

$i\%2$ es el resto de dividir i entre dos, que será cero si es par.

Recuerda que para comparar usamos el doble igual ==

Si se cumple, lo escribe, si no nada, y pasaría a la iteración siguiente, porque no hay más instrucciones.

Corto, sencillo y rápido. Para eso queremos lo bucles.

Hagamos otro ejemplo, **un programa que calcula potencias**.

Sería algo así como, me dices que la base vale 2, que el exponente vale 5 y te devuelvo un treinta y dos.

Para hacerlo, tengo que ir multiplicando dos por dos, me sale cuatro, luego otra vez por dos... hasta hacerlo cinco veces.

¿Cuántas variables necesito?

Necesito tener guardada la base, para ir tirando del 2 que tengo que usar para multiplicar una y otra vez.

Necesito tener guardado el exponente para saber cuándo he hecho el número de iteraciones suficiente y parar.

Necesito OTRA variable para ir guardando los números intermedios que me van saliendo. 4, 8, 16 y finalmente 32. Como no necesito quedarme con los resultados intermedios, puedo usar esta misma variable para almacenar el resultado final.

A esta variable la voy a llamar acumulado.

Probemos

```
'''Calcula potencias'''
print('Escribe la base')
base = int(input())
print('Escribe el exponente')
exponente = int(input())

acumulado = base

for i in range(exponente):
    acumulado = acumulado * base

print('El resultado es', acumulado)
```

Pongo base 2, exponente 5 y me sale... 64

Algo pasa, vamos a ver.

Si *acumulado* lo inicio con el valor de *base*, en la primera iteración tengo dos por dos... ya tengo cuatro. Así que en la primera iteración ya tengo dos al CUADRADO. Por lo tanto tengo que hacer una iteración menos. Probemos.

```
'''Calcula potencias'''
print('Escribe la base')
base = int(input())
print('Escribe el exponente')
exponente = int(input())

acumulado = base

for i in range(exponente-1):
    acumulado = acumulado * base

print('El resultado es', acumulado)
```

Perfecto, así funciona.

Otra variante sería, iniciar acumulado a uno y quizá así os parezca más “limpito”.

A mí me cuesta a veces ver a la primera donde tiene que empezar un contador o hasta donde tiene que ir, pero si probáis resulta fácil ajustarlo.

```
'''Calcula potencias'''
print('Escribe la base')
base = int(input())
print('Escribe el exponente')
exponente = int(input())

acumulado = 1

for i in range(exponente):
    acumulado= acumulado * base

print('El resultado es', acumulado)
```

Ojo cuando probéis los programas. He visto quien ha probado un programa para hacer potencias pidiéndole que hiciera DOS AL CUADRADO.

Esta prueba va a parecer que funciona aunque el programa intercambiara el papel de la base por el del exponente o aunque en lugar de hacer la exponenciación, simplemente, sume la base y el exponente. Vaya prueba...

Aunque es habitual que el contador de un bucle vaya desde cero, de uno en uno, hasta un máximo, no es obligatorio.

Si ponemos range(5) el contador va de cero a cuatro

Si ponemos range(3,7) el contador va desde 3 hasta 6

Si ponemos range(2,14,3) el contador va desde 2 hasta 13, saltando de tres en tres

Si ponemos range(14,2,-3) el contador va de 14 a 2, DECRECIENDO de tres en tres

Así que tenemos muchas posibilidades.

Aunque es muy usual numerar las iteraciones, también podemos ir recorriendo con nuestro “contador” cualquier cosa contable... por ejemplo, una palabra.

```
'''Cuenta las letras a que hay en una palabra'''  
print('Escribe una palabra en minúsculas y sin tildes')  
palabra = input()  
  
acumulado = 0  
  
for i in palabra:  
    if i == 'a':  
        acumulado= acumulado + 1  
  
print('El número de aes es', acumulado)
```

El “contador” `i` va recorriendo cada carácter de la palabra, y toma como valor ese carácter.

Si ponemos ‘javi’

En la primera iteración `i` toma el valor ‘j’, se pasa al condicional, como no es igual a ‘a’ pues no hacemos nada.

En la segunda iteración `i` toma el valor ‘a’, se pasa al condicional, como sí es igual a ‘a’ pues el acumulado se incrementa en una unidad.

Etcétera.

Finalmente imprimimos el valor acumulado y listo.

Esta forma es muy popular en Python por su sencillez, que es uno de sus principales valores y objetivos como lenguaje.

Queda más bonito aún si decimos `for letra in palabra`

Y podemos hacer lo mismo para cualquier “cosa contable”, dicho mejor, para cualquier **iterable**.

Hay muchas cosas que son iterables, que aún no conoces, pero por ejemplo los strings y las listas son iterables (uno por sus caracteres y el otro por sus elementos), así que no será raro ver:

```
for letra in palabra:
for elemento in mi_lista:
```

Y así nos despreocupamos de cuántos elementos tiene el iterable, sabiendo que los va a recorrer todos y parar al final. Esto funciona muy bien cuando no necesitas saber la posición que ocupa el elemento o el número de iteraciones que llevas, como en el caso de encontrar contar cuántas ‘a’ había en una palabra.

Cuando sí necesitas el “contador” de iteraciones puedes trabajar como hacíamos antes... o usar una interesante función **enumerate()**.

Esta función te dará los elementos del iterable, pero a la vez los “enumerará”, con lo que podrás usar ambos estilos a la vez. Es muy, muy útil.

Veamos un ejemplo:

```
palabra = input('Dime una palabra en minúsculas y sin tildes:
')
total = 0
posicion = []
for contador, letra in enumerate(palabra):
    if letra == 'a':
        total += 1
        posicion.append(contador + 1)

print('Número total de aes: ', total)
print('Posición de las aes: ', posicion)
```

He probado con la palabra cantigas y me ha salido lo siguiente

```
Dime una palabra en minúsculas y sin tildes: cantigas
```

```
Número total de aes: 2
```

```
Posición de las aes: [2, 7]
```

Veis que va estupendamente.

Analicemos el código.

Pido una palabra al usuario y la guardo en la variable tipo *string* palabra.

Inicializo un contador a cero (total)

Creo una lista vacía (posicion)

Y aquí viene la *madre del cordero*

```
for contador, letra in enumerate(palabra):
```

Esto crea DOS variables: contador y letra.

- La variable letra va a tomar en cada iteración el valor de un carácter de la palabra.
- La variable contador va a ir tomando valores cero, uno, dos... hasta que se acaben los elementos del iterable. ¡Y no necesitamos saber de antemano la longitud!

Ahora, dentro del bucle puedo usar uno u otro según me convenga.

Uso *letra* para comparar con la 'a' e ir sumando ocurrencias.

Uso *contador* para controlar en qué posición se da esa ocurrencia. En el ejemplo que hicimos antes con *for letra in palabra* eso no se podía hacer.

Como ves, con *enumerate*, tenemos lo mejor de los dos mundos.

¿Cómo sería el programa sólo usando índices?

```
palabra = input('Dime una palabra en minúsculas y sin tildes: ')
total = 0
posicion = []
for contador in range(len(palabra)):
    if palabra[contador] == 'a':
        total += 1
        posicion.append(contador + 1)

print('Número total de aes: ', total)
print('Posición de las aes: ', posicion)
```


Más enrevesado... aunque también puede hacerse.

Primero, tengo que averiguar cómo de larga es la palabra para que **range** llegue hasta allí y luego tengo que referenciar cada elemento de palabra por su índice palabra[contador].

Lo “pythonista”, (pythonic) es ir a lo más sencillo:

- Si con **elemento in iterable** es suficiente, pues así.
- Si no, **indice, elemento in enumerate(iterable)**

Para terminar, un recordatorio de algo que hemos usado así como si fuera una “idea feliz”, pero que es una práctica muy común. El uso de **contadores y acumuladores**.

Contadores: nos dan una idea de la iteración en la que estamos

Acumuladores: van actualizándose según algún criterio. Típicamente, si van *sumando los iniciaremos en cero*, si van *a ir multiplicando, los iniciaremos en 1*.

BUCLES ANIDADOS

Imagina que tengo una clase con 3 filas y 5 columnas. Tengo las notas de esos estudiantes organizadas como una lista de listas.

```
notas = [[6,6,7,10,9],[10,7,6,7,7],[5,5,10,7,10]]
```

El primer elemento de la lista **notas** es una lista con las calificaciones de la primera fila, el segundo elemento con las de la segunda fila, etc.

Si quisiéramos recorrer esta clase lo haríamos así:

Primera fila

 Miro al estudiante de la primera columna

 Miro al estudiante de la segunda columna

 ...

Segunda fila

 Miro al estudiante de la primera columna

 Miro al estudiante de la segunda columna

 ...

Supongo que ya te das cuenta de que necesitamos dos índices, dos contadores. Uno que nos indique en qué fila estamos y otro que nos diga en qué columna estamos.

Empecemos a ver cómo se puede recorrer la lista completa de manera ordenada con dos bucles anidados

```
'''Recorrer una lista de listas con dos bucles anidados'''
```

```
notas = [[6,6,7,10,9],[10,7,6,7,7],[5,5,10,7,10]]
```

```
for i in range(3):  
    for j in range(5):  
        print(i,j)
```

Este programa funciona así:

Primera iteración del primer bucle, la variable *i* vale cero y, dentro de esta primera iteración, el segundo bucle hace sus cinco iteraciones. Así que sale por pantalla

0 0

0 1

0 2

0 3

0 4

Y así termina la primera iteración del primer bucle.

Comienza la segunda iteración del primer bucle, la variable *i* vale 1 y, dentro de esa segunda iteración, el segundo bloque hace sus cinco iteraciones:

1 0

1 1

1 2

1 3

1 4

Y así termina la segunda iteración del primer bucle... y a por la tercera fila.

Ahora que hemos visto que los dos bucles nos recorren bien toda el aula, vamos a buscar, por ejemplo, los que tienen un diez.

```
'''Recorrer una lista de listas con dos bucles anidados'''
```

```
notas=[[6,6,7,10,9],[10,7,6,7,7],[5,5,10,7,10]]
for i in range(3):
    for j in range(5):
        # print(i,j)
        if notas[i][j] == 10:
            print(i,j)
```

Como ves he “desactivado” el print que teníamos antes comentándolo (para recordarte cómo se hace) y he añadido un condicional. Cuando la nota que se encuentre sea igual a 10 nos imprimirá las “coordenadas” *i,j* del estudiante en cuestión. Comprueba que funciona y no olvides que las listas se numeran empezando por cero.

BUCLES CONDICIONALES

En los ejemplos anteriores conocíamos el número de iteraciones, pero no siempre es el caso.

Cuando uno friega no piensa en

Repetir 5 veces:

```
Enjabonar()  
Aclarar()  
Secar()
```

Porque siempre llega alguien que trae algún plato nuevo... El bucle de fregado se parece más a:

Repetir HASTA que platos == 0:

```
Enjabonar()  
Aclarar()  
Secar()  
platos = platos-1
```

O también podría ser:

Repetir MIENTRAS que platos > 0:

```
Enjabonar()  
Aclarar()  
Secar()  
platos = platos - 1
```

Esto es lo que se denomina un bucle condicional. Antes de cada iteración se comprueba si se cumple la condición, en el caso afirmativo se realiza una nueva iteración, en caso contrario, se sale del bucle.

Pongamos un ejemplo sencillo, **intenta adivinar un número... hasta que aciertes**. También podemos decir: **MIENTRAS** falles.

```
'''Pide al usuario que adivine un número usando un while'''

clave = 2
prueba = 0 #por si hay en la memoria algún valor de una
ejecución anterior, inicializamos siempre las variables
print('Empezamos!')

while prueba != clave:
    print('Adivina un número entre 0 y 10')
    prueba = int(input())

print('Acertaste!')
```

El bucle condicional empieza con **“while”** que significa **mientras**, se añade la condición que te permite seguir iterando (en este caso que la prueba que escribe en usuario sea distinta a la clave).

Cuando esa condición deje de cumplirse, salimos del bucle y ejecutamos el resto del programa, en este caso, salida por pantalla “Acertaste!”.

Algo muy importante en los bucles condicionales es que **dentro del bloque de código del condicional SE CAMBIE LA VARIABLE SOBRE LA QUE SE INSPECCIONA LA CONDICIÓN**. Si no lo hacemos así, ¿cómo podría dejar de cumplirse la condición y cómo podríamos salir del bucle? Nos quedaríamos “embuclados” ejecutando iteraciones infinitamente o hasta que algo fallara (por ejemplo, un número que anduviera creciendo y que desbordase la memoria).

Objetivo cumplido

- Bucles con número de repeticiones
- Uso de range(fin)
- Uso de range(inicio, fin, paso)
- Variables “contador” y variables “acumulador”
- Usar otros elementos contables como iteradores (for i in string)
- Uso de *enumerate()*
- Bucles anidados
- Bucles condicionales (while)

BREAK Y CONTINUE. Interrumpiendo bucles.

A veces estamos recorriendo un bucle “buscando” algo, de manera que cuando lo encontramos ya no queremos seguir haciendo iteraciones, queremos “interrumpir” la búsqueda de alguna manera..

BREAK te saca del bucle de forma inmediata dejando sin hacer el resto de iteraciones
CONTINUE interrumpe la iteración en la que estás y pasa a la siguiente iteración.

Vamos a poner unos códigos de prueba sólo para ver cómo funcionan y luego hacemos unos ejemplos.

Ejemplo1.

```
#Al cumplirse la condición, continue hace que se interrumpa la
iteración y se pase a la siguiente, por eso no imprime 3
for i in range(1,10):
    if i==3:
        continue
    print (i)
```

Salida por pantalla:

```
1
2
4
5
6
7
8
9
```

Como ves al estar CONTINUE antes de print(i) nos salimos de la iteración i=3 antes de imprimir ese valor

Ejemplo2

#Al cumplirse la condición, break hace que se interrumpa el bucle entero y por eso no se imprimen números por encima de 22

```
for i in range(21,30):  
    if i==23:  
        break  
    print (i)
```

Salida por pantalla:

21

22

La instrucción BREAK interrumpe abruptamente el bucle y no se llevan a cabo más iteraciones. Como está antes de print(i) tampoco sale por pantalla el número 23.

Ejemplo3

#al haber dos bucles anidados, el continue del bucle interior solo termina la iteración en ese bucle, así que el efecto es que se pierde el 83

```
for i in range(51,60):
    if i==53:
        for j in range(81,90):
            if (j==83):
                continue
            print (j)
        print (i)
```

Salida por pantalla:

```
51
52
81
82
84
85
86
87
88
89
53
54
55
56
57
58
59
```

Veamos qué ha pasado aquí.

Primera iteración i=51, no se cumple el condicional, así que vamos a la línea siguiente FUERA del condicional, que es print(i), por lo que saca por pantalla un 51.

En la segunda iteración pasa lo mismo, saca un 52.

En la tercera iteración sí se cumple el condicional, así que nos pasamos a su código y nos encontramos un bucle. En ese momento, i se queda con el valor que tiene y empieza a variar la j, si no es igual a 83 no se cumple el condicional y la siguiente instrucción es print(j), por eso salen 81 y 82, pero cuando j es igual a 83 aparece un CONTINUE, así que interrumpe esa iteración (como está antes de print(j) el 83 no se imprime), pero pasa a la iteración siguiente lo que hace imprimir el 84 (ya no se

cumple el condicional), el 85... hasta el 89. Ahí terminamos el **for** y nuestra siguiente instrucción es `print(i)` y te recuerdo que estamos aún en la iteración `i=53`, por eso después del 89, aparece el 53 y el resto de números hasta que se termina el bucle en `i`.

Ejemplo4

#Al haber dos bucles anidados y usar un `break` en el bucle interior, sólo se interrumpe este y sigue con el bucle primario

```
for i in range(51,60):  
    if i==53:  
        for j in range(81,90):  
            if (j==83):  
                break  
            print (j)  
        print (i)
```

Salida por pantalla:

```
51  
52  
81  
82  
53  
54  
55  
56  
57  
58  
59
```

En este caso, la instrucción `BREAK` interrumpe el bucle en `j`, así que sólo puede llegar hasta el 82, y seguimos luego con el bucle en `i`.

Pongamos ahora dos ejemplos “más realistas”.

Un programa que te pide una contraseña.

```
'''Te pide una contraseña en un bucle infinito hasta que la  
aciertas'''
```

```
password = 'pandereta'
```

```
while True:  
    print('Escribe la contraseña')  
    guess = input()  
    if guess == password:  
        break  
    else:  
        print('Esa no es')  
  
print('Estás dentro!')
```

“while True” es una manera de escribir un bucle infinito. Sería como decir “mientras que verdadero sea verdadero...” Funcionaría igual que “while 1=1” o cualquier expresión lógica que sepamos que siempre es verdadera.

Si aciertan la contraseña, entramos en el condicional y **break** nos saca del bucle while, de forma que ya empezamos a ejecutar el código que hay después, que es el print de “Estás dentro”.

Un programa que va calculando los números primos.

¿Cómo podemos construir la lista de los números primos? Con mucha paciencia...

Empezamos nuestra lista con el 2.

Cogemos el número siguiente, el 3, y lo dividimos entre los primos que tenemos en nuestra lista (en este caso el 2). Si sale resto cero con alguno es que no es primo, y si no sale resto cero y llegamos a un punto en el que el cociente es menor que el divisor, entonces sabemos que es primo. El algoritmo que nos enseñaron en el cole.

Ahí veréis que hay dos puntos de ruptura, uno si encontramos que no es primo y otro si después de dividir el cociente es menor que el divisor.

Por ejemplo, imagina que vamos llevamos la lista llena hasta el 13, esto quiere decir que será [2,3,5,7,11,13].

Nos toca probar el 14. Hacemos la división 14 entre 2, resto cero. Vaya, no es primo. BREAK.

Nos toca probar el 15. Hacemos la división 15 entre 2, resto uno, cociente 7, mayor que 2, puede ser primo. Siguiendo primo de la lista, el 3. Hacemos la división 15 entre 3, vaya, resto cero. No es primo. BREAK.

Con el 16 pasa lo mismo que con el 14 (son pares). BREAK.

Nos toca ahora el 17. Hacemos la división 17 entre 2, resto uno, cociente 8, mayor que 2, puede ser primo, siguiente primo de la lista, el 3. 17 entre 3, resto 2, cociente 5, mayor que tres, puede ser primo. Siguiendo primo de la lista, el 7, resto 3, cociente 2 MENOR que 7... es primo! Lo añadimos a la lista de primos y BREAK.

#Vamos calculando los números primos y metiéndolos en una lista

```
limite = int(input('Dime el número máximo: '))
primos = [2]
for i in range (2, limite + 1):
    for j in primos:
        if i % j == 0:
            break
        elif (i//j) < j):
            primos.append(i)
            break
print(primos)
```

Te explico línea a línea.

Pedimos un límite. Aprovecho para recordarte la manera de resumir las dos líneas que solemos poner para recabar datos del usuario en una sola, poniendo como argumento en el INPUT el mensaje de texto que queremos dar al usuario.

Inicializamos la lista de primos con el número 2.

El bucle en i recorrerá todos los números de uno en uno hasta el máximo que nos han indicado.

El bucle en j recorre todos los primos que llevemos encontrados para cada número que vamos eligiendo en el bucle en i

Comprueba si el resto de dividir el número i entre el primo j da cero. Si es así, salimos del bucle en j, no hay que comprobar con más primos, ya sabemos que no lo es.

Si no sale cero (**elif**), comprueba si el cociente es menor que el divisor, en ese caso, sabemos que es primo, lo añadimos a la lista de primos y podemos dejar de comprobar con el resto de primos (bucle en j).

Finalmente, cuando ya hemos terminado con todos los números hasta el que nos dijeron, cuando se termine el bucle en i, podemos imprimir la lista completa de primos.

Programa que te pregunta el nombre y sólo si eres el indicado te pregunta la contraseña (adaptado de Automate the boring stuff with Python)

'''Te pregunta quién eres y sólo si eres la persona indicada te pregunta la contraseña.

Tomado de Automate the boring stuff con Python'''

```
while True:
    print('¿Quién eres?')
    nombre = input()
    if nombre != 'Javi':
        continue
    print('Hola, Javi. Dime la contraseña')
    password = input()
    if password == 'padentro':
        break
print('Estás dentro!')
```

Como antes, tenemos un bucle infinito del que sólo podremos salir *a las bravas*.

Nos pregunta quién somos y recoge la respuesta.

- Si no soy Javi, termina la iteración por el *continue* (no me preguntará la contraseña) y lo vuelve a empezar, preguntándome de nuevo quién soy.
- Si soy Javi sigue adelante y me pregunta la contraseña, que guarda en una variable.
- Si la contraseña es correcta sale del bucle infinito y me dirá “Estás dentro”, si no, vuelve a empezar el bucle y me preguntará el nombre de nuevo.

Objetivo cumplido

- Uso de break y continue
- Bucles infinitos

FUNCIONES. Los conjuros.

Ya hablamos antes de las funciones.

Son como conjuros. Tú las pronuncias y ALGO PASA.

Reconocemos que es una función porque tiene paréntesis a continuación de su nombre.

Ya hemos usado unas cuantas. De hecho fue lo primero que hicimos: Usar **print()**.

También conocemos **input()**, **len()**, **int()**⁷

Lo que vamos a hacer ahora es CREAR NUESTRAS PROPIAS FUNCIONES.

Hay que empezar recordando un par de cosas.

PARÁMETROS: Son valores/referencias que hay que pasar a la función para que funcione y van dentro del paréntesis. Por ejemplo, la función PRINT tiene como parámetros lo que queramos que saque por pantalla. Una función puede no llevar parámetros, pero hay que escribir los paréntesis

CON RETORNO O SIN RETORNO: Ya lo mencionamos antes. Una función sin retorno no entrega ningún valor⁸, simplemente hace una cosa y listo (como le pasa a PRINT), en cambio, **una función con retorno puede hacer cosas, pero también entrega un valor que debe recogerse en una variable** u otra estructura de datos. Así que las llamadas a funciones serán diferentes según tengan retorno o no.

Ejemplo SIN retorno: `print('¿Quién eres?')`

Ejemplo CON retorno: `edad = input('¿Cuántos años tienes?')`

La **gracia de usar funciones es ENCAPSULAR un bloque de código** de manera que el programa sea más claro y sencillo.

⁷ Aprovecho para decirte que también existe la función `str()` que convierte un entero en una cadena de caracteres, lo contrario que `int()`.

⁸ En realidad devuelve `None`, pero podemos ignorarlo en el alcance de este manual.

Hagamos una función que sume.

```
'''Función que suma dos números'''
```

```
def sumador(a,b):  
    resultado = a + b  
    return resultado
```

Las funciones se definen así:

Def

Nombre de la función

Paréntesis, con parámetros o vacíos

Dos puntos, porque viene un bloque de código

Indentación

Bloque de código de la función

return y el nombre de los valores que se devuelven

a, b simbolizan las dos variables que les pasaré, que no se llamarán necesariamente así (podrían ser edad1 y edad2, por ejemplo), es sólo para saber que ahí van a ponerse dos variables y lo que se les va a hacer luego, una vez dentro de la función.

Si ejecutáis el programa veréis que... no veis nada. **Este programa no hace nada.**

En el programa sólo está la definición de la función, **pero no se la ha invocado** ninguna vez. Sería como tener un libro de conjuros pero no haber dicho ninguno.

La definición no implica que se lleve a cabo, hay que llamarla explícitamente, y se le puede llamar tantas veces como se quiera. Recordad, **el programa hace lo que le dices y sólo lo que le dices.**

Vamos en serio ahora

```
'''Función que suma dos números'''

def sumador(a, b):
    resultado = a + b
    return resultado

var1 = int(input('dime el primer sumando '))
var2 = int(input('dime el segundo sumando '))

solucion = sumador(var1, var2)

print('El resultado de sumarlos es', solucion)
```

¿Has visto que he vuelto a usar el truco de poner el texto informativo en el argumento de la función `input()`?

Llamamos a la función que hemos creado y, como es una función con retorno, sabemos que **va a devolver un valor que TIENE que caer en algún sitio**, dentro de una variable, por ejemplo.

Por eso la llamada es `solucion = sumador(var1, var2)`

Y NO SE PUEDE LLAMAR A LA FUNCIÓN ASÍ: `sumador(var1, var2)`

Como ves a y b eran solo para identificar que había que pasarle dos parámetros a la función y lo que luego iba a hacer con ellos. Aquí hemos invocado a la función con los valores de var1 y var2 para los parámetros, y luego estos valores toman los papeles que tenían a y b en la definición.

Algo sobre lo que quiero que reflexionemos es sobre la variable resultado, a la que asignamos valor dentro de la función.

Añade una línea al final del programa para que imprima el valor de resultado... a ver qué pasa.


```
print(resultado)
```

A mí me aparece el siguiente error

```
Traceback (most recent call last):  
  File "main.py", line 12, in <module>  
    print(resultado)  
NameError: name 'resultado' is not defined
```

¿¿Que la variable resultado no está definida?? ¿¿Cómo que no?? ¡¡La he dado valor dentro de la función!!

Queridos, ya sabéis que lo que pasa en Las Vegas, se queda en Las Vegas.

Lo que pasa en una función, se queda en la función... (o casi)

resultado es lo que llamamos una **VARIABLE LOCAL** existe dentro de la función y sólo dentro de la función. A las variables que existen fuera de las funciones las llamamos **VARIABLES GLOBALES** y se puede acceder a su valor tanto dentro como fuera de las funciones.

Es cierto que devolvemos su valor al terminar la función, pero este valor se almacena en la variable global de destino, que es **solucion**.

Hagámoslo más limpiamente, para verlo claro.

Ejemplo1 **variable local que tiene valor dentro de la función y fuera no.**

```
def prueba():  
    var1 = 77  
    print('Valor de var1 DENTRO de la función', var1)
```

```
prueba()  
print('Valor de var1 FUERA de la función', var1)
```

Salida por pantalla

Valor de var1 DENTRO de la función 77

```
Traceback (most recent call last):  
  File "main.py", line 6, in <module>  
    print('Valor de var1 FUERA de la función', var1)  
NameError: name 'var1' is not defined
```

Mirad la última línea del error, var1 NO EXISTE fuera de la función.

Ejemplo2 **variable global cuyo valor puede usarse dentro de una función**

```
def prueba() :  
    var1=77+var2  
    print('Valor de var1 DENTRO de la función', var1)  
  
var2=3  
prueba()
```

Salida por pantalla

Valor de var1 DENTRO de la función 80

Por lo tanto, podemos usar los valores de las variables globales dentro de las funciones.

Ejemplo3 **NO PODEMOS** cambiar el valor de una variable global (por las bravas) desde dentro de una función

```
def prueba() :  
    var1=45  
  
var1=3  
prueba()  
  
print('Valor de var1', var1)
```

Salida por pantalla

Valor de var1 3

Fíjate en el programa, damos valor a var1, luego llamamos a la función, que suponíamos que le iba a cambiar el valor a 45, pero cuando imprimimos vemos que no. ¿Qué ha pasado?

Lo que ha pasado es que se ha creado una variable local con el mismo nombre que la global y que dentro de la función sí que vale 45, pero no nos ha dejado cambiar la variable global. Para hacerlo, veamos el ejemplo siguiente.

Ejemplo4 Cambiamos el valor de una variable global desde DENTRO de una función

Aquí te cuento como hacerlo, usando **global**.

```
def prueba():  
    global var1  
    var1=45  
  
var1=3  
prueba()  
  
print('Valor de var1', var1)
```

Salida por pantalla

Valor de var1 45

global var1 le indica a la función que no debe crear una variable local con el nombre var1 porque cuando pongamos var1 nos estaremos refiriendo a la variable global var1, así que ya podemos manipularla desde la función.

Esto es un asunto peliagudo y no está recomendado. Recuerda que una función está pensada para encapsular un bloque de código y “despreocuparnos” un poco. Funciona un poco como una caja negra, **solo vemos lo que entra y lo que sale**. Recuerda también que esa función podría estar programada por otra persona y que no nos ponemos a mirar cómo son sus tripas, sólo usarla. **Si las funciones anduvieran manipulando variables globales, podrían producir errores sin que ni siquiera supiésemos de dónde vendrían.**

IMPORTANTE: Por lo anterior, **se recomienda que la interacción de las funciones con el resto del programa se haga a través de los parámetros para la entrada y return para la salida de información.**

Objetivo cumplido

- Qué es una función
- Funciones con parámetros
- Funciones con retorno (return)
- Variables locales y globales

LIBRERÍAS. Alguien ya ha lavado los platos...

Como os contaba, las funciones son bloques de código autónomos que hacen tareas concretas y que podemos llamar desde nuestros programas.

Podemos programarlas nosotros o podemos usar las que hayan hecho otros.

Nunca perdáis de vista que no somos pioneros en casi nada, por lo que nuestros problemas ya los han tenido otros y puede que los hayan solucionado.

Las funciones se agrupan en librerías que suelen ser temáticas (estadística, p.ej.)

Para poder usar una función que no hemos programado nosotros, tenemos que **IMPORTARLA**.

Podemos traernos la función sola o bien la librería completa, si queremos usar más de una.

Se hace con el comando **import**

Por ejemplo, hay una librería para cosas relacionadas con números aleatorios que se llama **random**

Si queremos usar funciones de esa librería empezaremos nuestro programa con la línea

```
import random
```

Después sólo tendremos que llamar a la función que queramos con los parámetros adecuados.

¿Cómo sabemos esto? Pues hay que buscar la información, no se sabe por ciencia infusa.

[Aquí tenéis algunos módulos](#) de la librería estándar de Python, pero si buscáis por la red encontraréis [referencias a otras librerías muy populares](#), como Numpy, Scrapy, etc.

OJO: Importar una librería, al igual que definir una función, no implica ejecutarla, simplemente está ahí disponible.

Permíteme una metáfora:

Es parecido a que tú importaras la librería “Cosas de casa” a tu cerebro.

Ahí tienes disponible las funciones: `freir_huevo()`, `lavar_ropa()`, `barrer()`, etc.

Pero no se llevarán a cabo hasta que no se invoquen, una vez, mil veces o ninguna.

Quizá hoy tu día sea:

```
import cosas_de_casa

barrer()
lavar_ropa(10 prendas)
for i in range (len(invitados)):
    freir_huevos()
echar_de_casa(invitados)
```

Vamos con un ejemplo real.

En la librería **random** existe una función que se llama **randint()** que te devuelve un número entero aleatorio entre dos valores que le especifiques.

Hagamos un programa para que se adivine un número y aprovechamos para practicar todas las cosas que sabemos ya.

```
import random

intentos = 0
maximo = 0
clave = 0
prueba = 0
intentoMax = int(input('Cuántos intentos quieres? '))
maximo = int(input('¿Hasta qué número máximo puedo elegir?: '))
clave = random.randint(1, maximo)
while True:
    intentos += 1
    if (intentos > intentoMax):
        print('Te quedaste sin intentos')
        break
    prueba = int(input('Intenta adivinarlo: '))
    if (prueba == clave):
        print('Acertaste en ' + str(intentos) + ' intentos')
        break
    elif (prueba < clave):
        print('Te quedaste corto y te quedan ' + str(intentoMax
- intentos) + ' intentos')
    elif (prueba > clave):
        print('Te pasaste un poco y te quedan ' + str(intentoMax
- intentos) + ' intentos')
```

Te he puesto en negrita las dos líneas donde se aplica el tema de las librerías, pero voy línea por línea.

Importamos la librería **random** para poder usar alguna de sus funciones
Iniciamos las variables **intentos**, **maximo**, **clave**, **prueba** a cero.

Le pedimos al usuario que nos especifique el número máximo hasta el que podemos elegir.

Ahora, la *chicha*:

```
clave = random.randint(1,maximo)
```

La función es `randint` (del inglés *random integer*) así que nos dará un entero al azar. Límite inferior 1, límite superior el valor que nos dio el usuario y que guardamos en la variable **maximo**

random.randint() la primera parte nos dice que la función es del módulo **random**

Como hemos leído en la ayuda o visto en ejemplos que es una función con retorno, la llamamos de forma que el valor que devuelve caiga dentro de una variable, en nuestro caso **clave** que va a ser el número que queremos adivinar.

Lo que sigue es un bucle infinito del que saldremos bien porque adivinemos la clave, bien porque se nos acaben los intentos.

Al principio de cada iteración aumentamos intentos en una unidad (recuerda que podía escribirse de esta forma `x+=1`)

Si el número de intentos es mayor que `intentoMax`, nos dice que hemos perdido y sale fuera del bucle.

En la siguiente línea (fuera de ese primer **if**) me pide que intente adivinarlo y lo guarda en prueba.

Y ahora usando **if** y **elif** vemos si hemos acertado (y ahí se sale del programa), o bien nos hemos quedado cortos o largos y, en estos casos, nos informa antes de volver a la siguiente iteración del bucle `while`.

Hemos usado es otra forma de sacar texto y variables con un `print`

```
print('Te pasaste un poco y te quedan' +  
      str(intentoMax-intentos) + ' intentos')
```

Aquí lo que hemos hecho es convertir el número entero (`intentoMax - intentos`) a una cadena de caracteres (`string`) y luego “sumar” cadenas de caracteres, lo que llamamos **concatenar**.

Es la operación contraria que hacíamos con `int()`. Al igual que `int('9')` te devolvía el número nueve, `str(9)` te devuelve la cadena `'9'`

Aquí un ejemplo donde se calcula el valor del número pi, según el Método de Montecarlo. Te lo explico.

Imagina que tenemos un papel cuadrado y dibujamos un círculo inscrito.

Ahora disparamos aleatoriamente puntos en ese papel.

La probabilidad de que el punto esté en el círculo es proporcional al área del círculo.

Así que el cociente entre el número de disparos dentro del círculo y el número de disparos totales será “parecido” al cociente entre las áreas del cuadrado y del círculo. Ese “parecido” lo será más cuanto mayor sea el número de disparos que hagamos.

Así que:

disparos dentro del círculo/disparos totales = área círculo/área cuadrado

Área del círculo $\pi \cdot r^2$

Consideramos un círculo de radio 1 y sólo la cuarta parte del dibujo.

Luego, el área de nuestro círculo será $\pi/4$
 el área del cuadrado que nos queda será 1

Concluimos que

$\text{Disparosdentro/disparostotales} = \pi/4$

Por lo que, una estimación de pi será, despejando.

$\pi = 4 \cdot \text{disparosdentro/disparostotales}$

Para saber si un disparo está dentro del círculo, su distancia al centro: $\sqrt{x^2+y^2}$ debe ser menor que el radio (1).

La verdad es que podíamos elevar las dos cosas al cuadrado y nos evitábamos hacer la raíz cuadrada, pero así te puedo mostrar una función del módulo math: **math.sqrt()**

Al final del programa mostramos el error de la estimación respecto del valor real.


```
import random
import math

numeroMaximo = int(input('Cuántos puntos quieres usar? '))

dentroCirculo = 0

for i in range(numeroMaximo):
    x = random.random() #genera un float entre 0 y 1
    y = random.random() #genera un float entre 0 y 1
    r = math.sqrt(x**2 + y**2)
    if (r < 1):
        dentroCirculo += 1
#si la distancia al centro es menor que 1 el punto está dentro
del círculo
resultado = 4 * (dentroCirculo / numeroMaximo)
print(resultado)
errorAbs = resultado - math.pi #math.pi te da el valor de pi
errorRel = abs((errorAbs / math.pi) * 100)
print('El error es del ' + str(round(errorRel, 2)) + ' por
ciento')
#round redondea el valor del error relativo a dos decimales
```

Objetivo cumplido

- Qué es una librería
- Cómo importar una librería
- Cómo usar las funciones o constantes que contiene

MANEJO DE ERRORES. Cuenta con los fallos

En Python a los errores los llamamos **EXCEPCIONES**.

Antiguamente, cuando en un programa se producía un error, el programa se “colgaba”, se interrumpía bruscamente. Los errores podían ser muy variados: se había producido una división entre cero, se había intentado acceder a un archivo que no existía, etc.

En muchos lenguajes modernos tenemos la posibilidad de contar con que pueden producirse esos errores y hacer que el programa responda inteligentemente a ellos, sin quedarse colgado.

Programa que calcula el reparto de dinero para comprar un regalo entre varias personas

Este sería el programa “base”

```
'''Calcula a cuánto tocamos para pagar un regalo y controla que el usuario introduzca como número de asistentes un número y no una cadena de caracteres'''
```

```
precio = int(input('¿Cuánto cuesta el regalo '))  
asistentes = int(input('¿Cuántos sois a repartir '))
```

```
pagoIndividual = precio/asistentes
```

```
print('Tocáis a ', pagoIndividual, 'euros')
```

Este programa puede dar varios errores

1. El precio no es un número
2. El número de asistentes no es un número
3. El número de asistentes es cero

Para manejar los errores se “envuelve” el código “peligroso” en un bloque precedido por **try** (vamos a INTENTAR ejecutar el código)

Hagamos “cascar” primero el programa para ver qué error nos sale y así sabremos cuál es la denominación exacta para poderlo “recoger”.

Ejecuta el programa y, cuando te pida el precio del regalo, escribe una palabra, yo voy a poner CROQUETAS.

Esto me sale por pantalla

ValueError: invalid literal for int() with base 10: 'croquetas'

En resumen, le fastidia que le haya pasado un string a `int()` que no sea interpretable como un número en base 10, como hubiera pasado con ‘42’, por ejemplo. Y el error se llama `ValueError`. Entendido.

Así que pondré el código peligroso dentro de un **try** y estaré preparado para la excepción `ValueError`. Algo así como

```
try:
    precio = int(input('¿Cuánto cuesta el regalo? '))
except ValueError:
    print('Eso no es un número')
```

Salida por pantalla

¿Cuánto cuesta el regalo? croquetas

Eso no es un número

¿Cuántos sois a repartir

Veis que ha visto que es un error y ha seguido adelante sin quedarse colgado. ¡Bien!

El problema es que no basta con que me informe de que he puesto mal el número, debería permitirme volverlo a poner bien. Quizá con un bucle infinito y un `break`...

```
while True:
    try:
        precio=int(input('¿Cuánto cuesta el regalo '))
        break
    except ValueError:
        print('Eso no es un número')
```

¡Así funciona! Si la operación no da error, se sale del bucle, pero si da error, te informa y vuelve a preguntarte.

Para asegurarnos de que el número de asistentes ES un número podemos hacer lo mismo... pero nos queda controlar que el número no sea cero.

Podíamos hacerlo usando nuestros conocimientos anteriores, con condicionales. Algo así como.

```
while True:
    try:
        asistentes = int(input('¿Cuántos sois a repartir '))
        if asistentes < 1:
            continue
        else:
            break
    except ValueError:
        print('Eso no es un número')
```

Si el número de asistentes es menor que uno (cero o negativo) paso a la siguiente iteración del bucle (continue), vaya, te lo pregunto otra vez.

Si es un número mayor que uno, me salgo del bucle.

Si no es un número y me da un error, te informo y paso a la siguiente iteración, te lo vuelvo a preguntar.

Pero vamos a dejar que “casque” al dividir por cero y manejar la excepción, para practicar. Así que volvemos al código anterior y añadimos esa nueva parte.

```
while True:
    try:
        precio = int(input('¿Cuánto cuesta el regalo '))
        break
    except ValueError:
        print('Eso no es un número')

while True:
    try:
        asistentes = int(input('¿Cuántos sois a repartir '))
        break
    except ValueError:
        print('Eso no es un número')

pagoIndividual = precio / asistentes

print('Tocáis a ', pagoIndividual, 'euros')
```

Si probáis a meter cero como número de asistentes, ahí no dará problemas, sino en la línea donde se hace la división.

Salida por pantalla:

¿Cuánto cuesta el regalo 22

¿Cuántos sois a repartir 0

Traceback (most recent call last):

File "main.py", line 18, in <module>

pagoIndividual=precio/asistentes

ZeroDivisionError: division by zero

Así que, aquí el código peligroso es la división, y será ese el que tengamos que meter en un **try** y recoger la excepción que nos indican `ZeroDivisionError`

```
try:
    pagoIndividual=precio/asistentes
    print('Tocáis a ', pagoIndividual, 'euros')
except ZeroDivisionError:
    print('Creo que venís muy pocos ;)')
```

Salida por pantalla

¿Cuánto cuesta el regalo 22

¿Cuántos sois a repartir 0

Creo que venís muy pocos ;)

Fíjate que he metido el **print** que decía a cuanto tocaban dentro del **try**, la razón es sencilla... al no poderse hacer la operación, si lo dejo fuera e intenta ejecutarse, me da un error de *NameError: name 'pagoIndividual' is not defined* porque no ha sido posible crear esa variable.

¿Y si usamos un sólo **try** y varios **except** para todo el tema de asistentes?

```
while True:
    try:
        precio=int(input('¿Cuánto cuesta el regalo '))
        break
    except ValueError:
        print('Eso no es un número')
while True:
    try:
        asistentes=int(input('¿Cuántos sois a repartir '))
        pagoIndividual=precio/asistentes
        break
    except ValueError:
        print('Eso no es un número')
    except ZeroDivisionError:
        print('Creo que venís muy pocos ;)')

print('Tocáis a ', pagoIndividual, 'euros')
```

Veréis que funciona estupendamente.

También lo podríamos meter todo en un **try** y recoger los dos tipos de errores que nos pueden suceder, el problema es que al volver al inicio del bucle, si el precio estaba bien metido, tendremos que volver a teclearlo, pero quedaría así, probadlo.

```
while True:
    try:
        precio = int(input('¿Cuánto cuesta el regalo '))
        asistentes = int(input('¿Cuántos sois a repartir '))
        pagoIndividual = precio / asistentes
        break
    except ValueError:
        print('Eso no es un número')
    except ZeroDivisionError:
        print('Creo que venís muy pocos ;)')

print('Tocáis a ', pagoIndividual, 'euros')
```

Esta es la salida por pantalla probando a tocar las narices ;)

```
¿Cuánto cuesta el regalo gfdgf
Eso no es un número
¿Cuánto cuesta el regalo 25
¿Cuántos sois a repartir dfgd
Eso no es un número
¿Cuánto cuesta el regalo 25
¿Cuántos sois a repartir 0
Creo que venís muy pocos ;)
¿Cuánto cuesta el regalo 25
¿Cuántos sois a repartir 2
Tocáis a 12.5 euros
```

Ten en cuenta siempre las ayudas, las sugerencias, la documentación...

<https://docs.python.org/3/library/exceptions.html>

Objetivo cumplido

- Qué son las excepciones (errores)
- Cómo usar try-except

Diccionarios. Otros tipos de datos

Una pregunta que nos hacemos antes de ponernos a “picar código” es: “¿Qué estructura tienen nuestros datos?”. ¿Serán enteros, cadenas de caracteres, listas...?

Aunque hemos hablado de algunos tipos de datos, no hemos agotado el tema, dejadme que os hable de alguno más.

A primera vista podría pareceros una lista, pero no.

```
mi_buga = {'marca': 'seat', 'modelo': 'panda', 'year': 1990}
```

Importante que os fijéis en:

- Las llaves, en lugar de paréntesis o corchetes
- Las “parejas” separadas por comas.

Aquí no hay un orden, primer elemento, segundo elemento.. NO hay un índice numérico. Hay parejas de **PALABRAS CLAVE y EL VALOR QUE TOMAN. (keys and values)**

Cuando queramos “extraer” el valor ‘seat’ deberemos preguntar por la clave ‘marca’, NO por el primer elemento.

Nos pueden servir para representar distintas características de un elemento, como en nuestro ejemplo del coche, y ahí veis la lógica de que esas características no tengan un orden natural.

También nos pueden servir para almacenar una misma característica para distintos elementos, por ejemplo, el cumpleaños de distintas personas.

```
cumples = {'Juan': '2 Marzo', 'María': '5 Junio', 'Luisa': '15  
Septiembre'}
```

¿Veis la diferencia con una lista?

Si hiciera una lista con los cumpleaños de amigos, debería saber qué número de orden corresponde a cada uno, cuando el orden es arbitrario. Aquí lo importante es, tener las fechas en una misma estructura (estoy guardando algo similar) y unidas cada una con el nombre correspondiente.

¿Por qué es esto potente?

Porque sólo necesito la clave para acceder al valor deseado.

Que quiero saber el modelo de mi coche

```
print(mi_buga['modelo'])
```

Salida por pantalla:

panda

O si quiero saber cuándo cumple años María, pediré

```
print(cumples['María'])
```

También puedo dar valores a las claves con una sencilla asignación, lo que podría crear una clave nueva si 'modelo' no existiera.

```
mi_buga['modelo']='renault'
```

O averiguar si hay una cierta clave de mi interés dentro de un diccionario

```
if 'modelo' in mi_buga:  
    print('modelo está en el diccionario')
```

Ya ves que **las claves cumplen la función que hacía el índice en las listas**, de manera que igual nos funciona hacer un bucle que recorra las claves... probemos.

```
for x in mi_buga:  
    print(x)
```

Salida por pantalla:

marca

modelo

year

Si escribimos esto:

```
for x in mi_buga:  
    print(mi_buga[x])
```

Salida por pantalla:

seat

panda

1990

Analicemos este **ejemplo de Automate the boring stuff with Python (cap 5)** para ver una manera sencilla de que el usuario rellene un diccionario en tiempo de ejecución.

```
while True:
    print('Enter a name: (blank to quit)')
    name = input()
    if name == "":
        break

    if name in birthdays:
        print(birthdays[name] + ' is the birthday of ' + name)
    else:
        print('I do not have birthday information for ' + name)
        print('What is their birthday?')
        bday = input()
        birthdays[name] = bday
        print('Birthday database updated.')
print(birthdays)
```

Todo está dentro de un bucle infinito (*while True*) Mientras “verdad sea verdadero”...

Toma un nombre.

Si no metemos ninguno, se sale del bucle

Si metemos uno, comprueba si ya está en el diccionario

- Si está te da la fecha de nacimiento.
- Si no, te dice que no está, te pide el cumpleaños y lo mete en bday, para luego emparejarlo con el nombre que le diste.
- Te da una salida de confirmación de valores guardados.

Pasa a pedir otro nombre.

La última línea la hemos añadido para que imprima el diccionario completo.

Por cierto **los diccionarios también son OBJETOS**, así que tienes a tu disposición un conjunto de funciones (métodos) que ya están programadas y sólo tienes que “invocar” para obtener lo que deseas

(tomada de https://www.w3schools.com/python/python_dictionaries.asp).

Method	Description
<code>clear()</code>	Removes all the elements from the dictionary
<code>copy()</code>	Returns a copy of the dictionary
<code>fromkeys()</code>	Returns a dictionary with the specified keys and values
<code>get()</code>	Returns the value of the specified key
<code>items()</code>	Returns a list containing the a tuple for each key value pair
<code>keys()</code>	Returns a list containing the dictionary's keys
<code>pop()</code>	Removes the element with the specified key
<code>popitem()</code>	Removes the last inserted key-value pair
<code>setdefault()</code>	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value
<code>update()</code>	Updates the dictionary with the specified key-value pairs
<code>values()</code>	Returns a list of all the values in the dictionary

Recuerda que los métodos se llaman con el nombre del objeto particular que has creado, un punto, y el método en cuestión con los parámetros adecuados.

Por ejemplo, usemos `pop()` para eliminar una clave (se va el valor también, claro). Como tengo el diccionario `mi_buga` en la memoria, tecleo directamente en la ventana de comandos.

A ver si está bien

`mi_buga`

`{'marca': 'seat', 'modelo': 'panda', 'year': 1990}`

Correcto

Voy a sacar la clave ‘marca’

`mi_buga.pop('marca')`

`'seat'`

Me devuelve al valor que tenía y entiendo que se la ha “cargado”.

Veamos

`mi_buga`

`{'modelo': 'panda', 'year': 1990}`

¡Funcionó!

`set.default()` también es un método interesante y en **Automate the boring stuff with Python** hacen un ejemplo potente y elegante. A ver si os gusta tanto como a mí.

```
message = 'It was a bright cold day in April, and the clocks were striking thirteen.'  
count = {}
```

```
for character in message:  
    count.setdefault(character, 0)  
    count[character] = count[character] + 1  
  
print(count)
```

Pone un mensaje, podéis cambiarlo para que sea el usuario el que meta la cadena de caracteres.

Inicia el diccionario **count**

Hace un bucle que recorre toda la cadena de caracteres, uno a uno. `character` será la variable que recorre la cadena y va tomando los valores de cada carácter que la componen. Así que empieza valiendo 'I', luego vale 't', etc.

Desde el objeto `count`, se llama al método `setdefault()` que actuará de una de estas dos formas:

1. Si ese carácter aún no se ha contado, crea la clave con ese carácter y lo pone a cero.
2. Si ese carácter ya existe como clave, lo deja como esté. Así que al ejecutar la línea siguiente seguirá añadiendo uno sin problema.

En la siguiente línea añade uno al valor de la clave correspondiente a ese carácter.

Al final imprime el recuento de caracteres.

Sin la línea con la llamada al método `setdefault()` al intentar sumar uno a una clave que aún no hubiésemos encontrado, daría un error.

Como os decía enumerar características de un elemento, o también, agrupar valores de la misma característica para varios elementos, son dos ejemplos de usos de diccionarios: (marca, modelo, color) para un coche, o años de fabricación para distintos coches.

Y, ¿podríamos pensar en hacer un diccionario de diccionarios, como cuando hicimos una lista de listas (y nos salía una tabla si sólo usábamos dos dimensiones)?

La verdad es que sí, podemos hacer un diccionario cuyas claves sean primer hijo, segundo hijo... y sus valores otros diccionarios como hacen en https://www.w3schools.com/python/python_dictionaries.asp

```
myfamily = {  
    "child1" : {  
        "name" : "Emil",  
        "year" : 2004  
    },  
    "child2" : {  
        "name" : "Tobias",  
        "year" : 2007  
    },  
    "child3" : {  
        "name" : "Linus",  
        "year" : 2011  
    }  
}
```

child1 es una clave del diccionario “grande” y su valor es el diccionario “pequeño” con las claves/valores name:Emil, year:2004

child2 es otra clave y child3 es otra clave, mientras que los diccionarios

`{"name": "Tobias", "year": 2007}` y `{"name": "Linus", "year": 2011}`

Son sus valores respectivos.

Algo interesante que también puedes observar en este ejemplo es cómo han usado varias líneas para definir el diccionario y que quedase mucho más claro.

Ya ves, no todas las series de números son “listas”.

PIENSA BIEN EN LAS CARACTERÍSTICAS DE TUS DATOS ANTES DE ELEGIR CÓMO ESTRUCTURARLOS, será muy importante a la hora de trabajar con ellos.

Objetivo cumplido

- Qué son los diccionarios
- Cómo acceder a las claves y los valores
- Cómo usar sus métodos o iterar sobre ellos

Acceso a ficheros

En nuestros programas previos nos conformábamos con entradas de teclado y salidas por pantalla, pero **la informática toma su mayor sentido al automatizar tareas con gran volumen de datos y, para eso, necesitamos poder escribir y leer archivos**. Esos archivos pueden proceder de la red (datos estadísticos públicos), de sensores que tienen un histórico de medidas, bases de datos, etc. De la misma forma, nuestra “salida” también podría ser la entrada de otro programa, en forma de fichero de datos.

Escribir o leer un archivo serían funciones que tendríamos que programar nosotros... de no ser por la orientación a objetos de la que ya hemos hablado⁹.

Así que, definiremos nuestro archivo como un objeto de tipo **file** (“archivo”) y eso nos habilitará para llamar a los métodos que ya han programado otros para que puedan actuar sobre esos archivos, como **read()** o **write()**.

Antes de empezar ya sabes que todos los sistemas operativos usan una estructura de carpetas en árbol, por lo tanto, lo primero sería saber en qué directorio (que es como antes llamábamos a las carpetas) estamos trabajando.

Según tengas Python instalado o estés trabajando en un editor online, esto puede cambiar, pero podemos “preguntarlo”.

Para eso necesitamos una función que nos indique el “current working directory” (la carpeta en la que estamos).

Esa función es: `os.getcwd()`

Fíjate en cómo se procura elegir un nombre que sea fácil de interpretar: `get` (conseguir) `cwd` (current working directory). Fácil de interpretar... en inglés, claro.

Si te llama la atención “**os.**”, recuerda que es la manera en la que decimos que una función está dentro de un módulo, de un conjunto de funciones que están agrupadas y que debemos importar para poder usar. En este caso, relacionadas con el sistema operativo -**operative system**-, se llama: Miscellaneous operating system interfaces.

⁹ Si acabas trabajando con programación orientada a objetos, te quejarás muchas veces de sus inconvenientes, así que, permíteme que te recuerde sus ventajas ;)

De esta forma, si escribo sólo `os.getcwd()` me dará un error porque la función no está, no ha sido importada. Por lo tanto, escribiremos:

```
import os
print(os.getcwd())
```

Y obtengo por pantalla
/home/runner

Si estuviérais usando una versión instalada de Python, os habría dado una carpeta de vuestro PC, pero, al leer esto... no tengo muy claro donde me va a dejar o a poner las cosas. Ya lo veremos ;)

Vamos a probar a crear un archivo a ver dónde aparece.

Para crear un archivo hay que usar la función `open('Nombre del archivo')`. Es una función preexistente, no hay que importar módulo alguno.

La función `open` DEVUELVE un objeto de tipo fichero, al que ya le podremos aplicar los métodos correspondientes para leer o escribir, por ejemplo.

Por lo tanto, *queríamos* poner

```
f = open('ficheroPrueba.txt')
```

Pero nos sale un error

FileNotFoundError: [Errno 2] No such file or directory: 'ficheroPrueba.txt'

Python entiende que queremos abrir un fichero que... aún no existe, para eso tenemos opciones en el método `open()`

'r' si queremos sólo leer (valor por defecto). Da error si el archivo no existe.

'x' crea un archivo. Da error si el archivo ya existe.

'w' permite escribir en un archivo. Sobreescribe el contenido anterior si existiera.

'a' permite añadir contenido a un archivo.

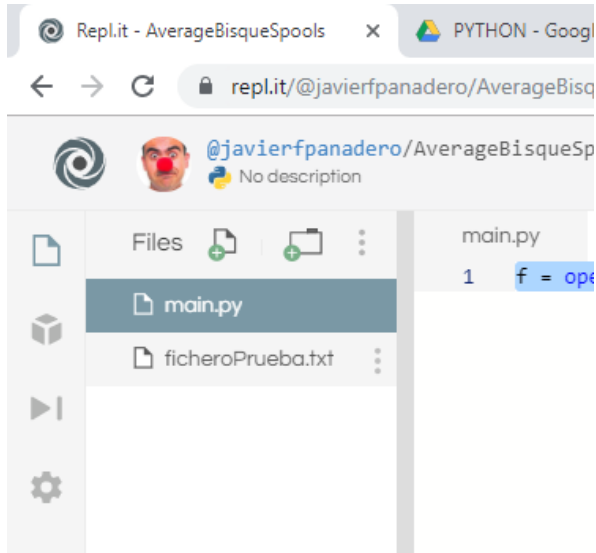
Los dos últimos valores (w,a) **crean** el archivo si este no existiera.

A ver si así...

```
f = open('ficheroPrueba.txt', 'w')
```

Pues Python no se queja... pero no veo que haya sucedido nada. Se supone que se ha creado un archivo, pero, ¿dónde está?

Mira un poco a tu izquierda... ¡Ajá!



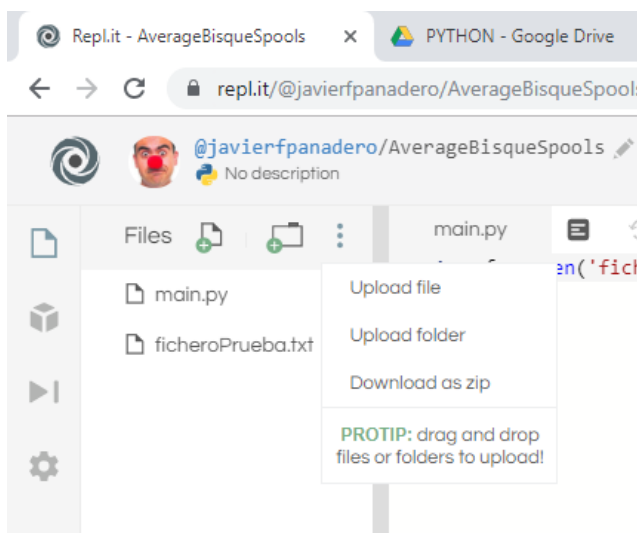
Puedes ver que Repl te ha creado un espacio para que tengas los ficheros (o los estructuras en carpetas si lo deseas).

De forma que, para el alcance de este tutorial podríamos leer y escribir archivos en este lugar y listo, pero, ¿podemos acceder al disco del ordenador y manipular archivos allí?

Parece que Repl actúa en plan “sandbox” (caja de arena), un lugar seguro para jugar. Si intento que me cree el fichero fuera, dándole la dirección completa (path) me crea un nuevo fichero, pero otra vez, en esta carpeta virtual.

NOTA: Quizá sea bueno tomarnos un segundo para ver que los programas que escribíamos en nuestro proyecto se guardaban en un fichero que se llamaba *main.py* (programa principal, hecho en Python). Podríamos hacer otros programas secundarios que se guardaran en esa misma carpeta y que pudiera llamarlos el programa principal, librerías u otros tipos de archivos.

Si hacéis click en los tres puntos de al lado del icono “carpeta nueva”, aparece lo siguiente:



Muy interesante.

Puedo subir los ficheros que quiera (con datos, por ejemplo)

Luego puedo descargarme los ficheros con los resultados grabados.

¡Estupendo!

Te habrás fijado antes que hemos nombrado el archivo como .txt, un archivo de texto plano, muy útil para evitar los códigos de formato que hay en archivos de Word o similar (tipos de letra, márgenes, etc.), pero este no es el único formato en el que se pueden guardar o escribir ficheros en Python. También puede hacerse en formato binario, pero para eso hay que especificarlo, ya que el formato texto (a base de cadenas de caracteres es el formato por defecto).

Así que si quisiéramos que lo que escribamos se entienda luego como una imagen, por ejemplo, podemos escribir en binario en ese fichero, especificando al lado del carácter de modo (r, x, w, a) un carácter para texto (t) o binario (b)

```
f = open('ficheroBinario', 'wb')
```

Empecemos...

El proceso para escribir o leer un archivo es el siguiente:

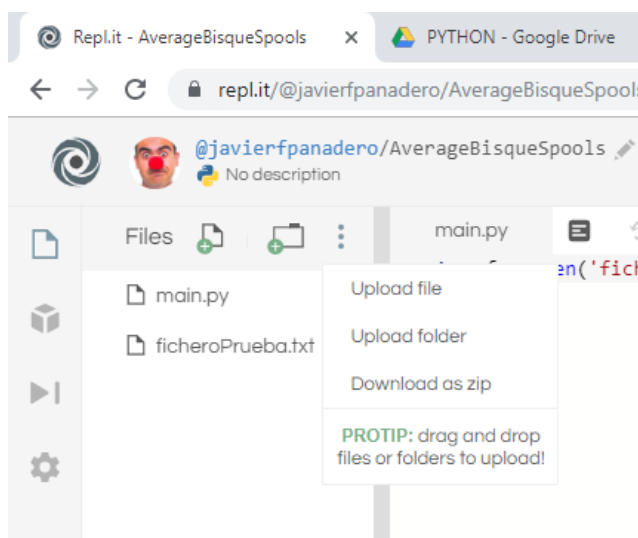
- Abrirlo
- Leer/escribir
- Cerrarlo

A los no iniciados nos suena raro eso de abrir un archivo, y que haya que cerrarlo luego, pero si piensas en que estas cosas se concretan en direcciones de memoria, contenido que se copia del disco duro a la RAM y que luego se actualiza, etc., seguro que ves que hay que seguir ciertos protocolos.

Leer un archivo.

Para leer un archivo, tenemos que tenerlo primero...

Abre el bloc de notas y escribe... Sería bonito que fuera 'Hello world!', ¿no?
Guárdalo en un sitio de acceso sencillo para subirlo a Repl.



Damos donde pone UPLOAD, seleccionamos el fichero y cuando aparezca ahí ya está subido.

Si no responde, recarga la página.

Recuerda. `open()` NO es un método, es una función que devuelve un objeto
El objeto es `f` (en nuestro caso)
Los métodos son: `read()`, `close()`

```
f = open('helloWorld.txt', 'rt') # modo lectura r tipo texto t
miSaludo = f.read()
print(miSaludo)
f.close()
```

Abrimos en modo lectura

Guardamos lo que se lea en una variable (`miSaludo`)

Imprimimos la variable (para ver que funciona)

Cerramos el archivo.

Me gustaría que pusieras en la ventana de comandos *miSaludo* para que vieras que te devuelve el valor entre comillas, señal de que lo está interpretando como una cadena de caracteres. MUCHO OJO a esto cuando queramos leer o escribir números, que tendríamos que usar `int()` o `str()`.

Si haces click al nombre del fichero que hemos subido, verás que te es posible editarlo desde allí mismo, en Repl.

Entonces, Panadero, ¿para qué nos haces escribirlo en el bloc de notas y subirlo si se podía crear allí y editar lo que fuera necesario?

Sencillo, para que tengas la seguridad de que podrías subir cualquier fichero de datos que te hubieses descargado de la red, que hubiese exportado cualquier programa (una hoja de cálculo, p.ej.) o que te hubiese entregado un sensor.

Uso del comando *with*

Esto de abrir y cerrar los archivos (`open` y `close`) da problemas... a la hora de cerrar. El primer problema es que, a veces se nos olvida incluir el cierre y ya la hemos liado, pero también sucede que puede saltar alguna excepción con el archivo abierto... Total, que la gente de Python ha buscado una solución para esto, de forma que el sistema va a controlar que siempre se cierre el archivo y además el código va a ser más simple. ¿No son majos?

Consiste en usar el comando ***with***.

Básicamente vamos a sustituir este tipo de código:

```
f = open('helloWorld.txt', 'rt')
miSaludo = f.read()
print(miSaludo)
f.close()
```

por este otro

```
with open('helloWorld.txt', 'rt') as f:
    miSaludo = f.read()
    print(miSaludo)
```

Funciona y nos olvidamos de tener que cerrar y el resto de excepciones. Sabemos que eso lo maneja un “*context manager*” y simplemente abrimos los archivos y hacemos con ellos lo que tengamos que hacer.

Fíjate, eso sí, que lo que hacemos está INDENTADO, que ***with*** crea un bloque de código donde vamos a llevar a cabo la lectura o escritura.

Si probáis lo que hicimos al principio usar sólo la primera sentencia para “crear” un archivo que ahora sería `with open('cosol.txt', 'w') as f:` veréis que da un error al no tener el bloque de código de después.

¡Vamos al lío!

Escribir un archivo

Vamos ahora a exportar datos que previamente ha calculado nuestro programa.

```
edad = int(input('Dime cuántos cumpleaños este año '))
year = 2022 - edad

with open('datosJavi.txt', 'wt') as f:
    f.write('Javi nació en el ' + str(year))
```

Recordamos que la entrada de teclado es un string, así que lo convertimos a entero. Calculo el año.

Abro el fichero para escribir texto.

Escribo una cadena que sale de concatenar (“empalmar”) la frase con el año CONVERTIDO en string, para poder escribirlo.

Si miráis a la izquierda en vuestro Repl, veréis que ha aparecido un nuevo archivo, haced click y podréis ver su contenido... saben cosas más(!!)

Escribir un archivo, añadiendo información

```
with open('secuenciaNum.txt', 'at') as f:
    while True:
        num = input('Escribe un número y pulsa intro. Sólo
intro para terminar')
        if num == '':
            break
        else:
            f.write(num)
```

Empezamos abriendo el archivo, pero en modo ‘append’ para que vaya añadiendo la información, no sobrescribiendo cada número que le demos.

Hago un bucle infinito del que saldremos cuando le demos a intro sin haber introducido ningún carácter.

Mientras, irá añadiendo los caracteres (no necesariamente números) que le vayamos dando. De hecho, los cogemos como strings del teclado, los escribimos como strings en el archivo.

Mira el contenido del archivo (haciendo un click) en su nombre en la parte izquierda de la pantalla de Repl y verás que ha ido añadiendo los números que hemos dicho, pero que no ha introducido ningún carácter entre esos números: ni espacios, ni comas, ni tabulaciones, ni cambios de línea...

¡Qué estupenda noticia!

Eso significa que nos deja toda la libertad para que añadamos nosotros los que queramos, o nada si no queremos, **así que tenemos el control total.**

Recuerda: Lo peor de la programación es que hace lo que le digo Y LO MEJOR ES QUE... HACE LO QUE LE DIGO.

No es difícil cambiarlo, Pruébalo.

Si quieres añadir una coma, simplemente concatena la cadena que te da el usuario con el carácter coma. `f.write(num + ',')`

Para tabulaciones escribe '\t' y para cambios de línea '\n'

Las tabulaciones, las comas y los cambios de línea son caracteres muy habituales para separar datos, así que debes conocerlos. Dos formatos de datos muy comunes son CSV y TSV (valores separados por comas y por tabulaciones, respectivamente). Incluso desde las conocidas hojas de cálculo pueden exportarse a esos formatos... así como importarse.

Leer partes del archivo

Imagina que sólo queremos leer partes algunos caracteres del archivo, entonces podemos **usar el parámetro del método read()** y poner dentro del paréntesis hasta que carácter queremos leer. El valor por defecto es -1 que simboliza el final de la cadena.

Por ejemplo, si cambiamos el programa ejemplo de lectura y en la línea que se llamaba al método escribimos `miSaludo = f.read(4)`, obtendremos en pantalla los cuatro primeros caracteres (Hell)

También podemos usar otro método que lee líneas completas. **`readlines()`**

OJO con este método:

Da como salida una LISTA, no una cadena con todos los caracteres del archivo.

Cada ÍTEM de la lista es UNA LÍNEA.

¿Lo probamos?

He cambiado el txt y he añadido otra línea.

He cambiado la línea de lectura por `miSaludo = f.readlines()`

Esto es lo que me sale por pantalla
['Hello World!\n', 'Passa con tu body?']

Ya veis que es una lista (notad los corchetes), **cada ítem es una línea entera** (con su cambio de línea y todo, que también es un carácter que hay en el fichero. Oye, no se le pasa una a Python)

A **readlines()** se le puede pasar un parámetro y es el número máximo de caracteres a leer (NO el número de líneas), de manera que, si va a leer la siguiente línea y ya ha alcanzado el máximo de caracteres que iba a leer, deja de leer.

Ahora lo que me pide el cuerpo es hacerte un ejemplo de lectura de un archivo CSV... algo así como:

- Abrir archivo.

- Iniciar bucle.

- Leer carácter

 - Si no es una coma

 - Añadirlo a una variable (e ir formando una “palabra”)

 - Si es una coma

 - Tomar la palabra que llevamos leída y meterla en una variable, lista, diccionario...

- Cerrar archivo

Pero, ¿sabes qué?

A estas alturas ya deberías tener en mente cierta actitud.

Esa necesidad tan razonable que tienes, tan relativamente fácil de implementar... ¿no estará ya implementada en algún objeto, en alguna librería?

Así que, aunque es un interesante ejercicio la implementación de ese programa que pueda leer CSV y guardar en un diccionario de diccionarios una tabla que fuera una base de datos de personas con distintos campos... (profes, alumnos curiosos... adelante), prefiero volver a hacer hincapié en que, si tienes un problema real y no es sólo un ejercicio para coger soltura... tu solución empieza por

```
import csv
```

Como podrás leer por aquí <https://realpython.com/python-csv/>

Objetivo cumplido

- Sabemos cómo llevar o sacar ficheros del “alcance” de Python
- Escribir y leer en ellos

Despedida:

Hasta aquí hemos llegado. Ha sido un largo viaje (para mí también), espero que os pueda servir, sobre todo a principiantes y profesores que podáis usar este manual bien como guía o bien para que vuestros alumnos aprendan de manera autónoma.

Agradecer los consejos de amigos, pero sobre todo a Al Sweigart, el autor de Automate the boring stuff with Python por su fantástico libro que ha sido guía e inspiración de este trabajo.

Después tenéis algunos ejemplos y aplicaciones más y al final algunas referencias, de entre las miles que hay por la red.

Vuelvo a recomendar el libro de Al Sweigart, no tanto por su profundidad (que la tiene) sino por su actitud ante la programación, entendiéndola como que puede resultar ser un “superpoder” que te permita (seas de ramas científicas o humanidades) automatizar tareas en este mundo con tanta información por procesar. Así que te aconsejo que no te pierdas la segunda mitad del libro, en la que podrás encontrar potentes aplicaciones una vez que domines las bases.

De nuevo, **si te ha servido el trabajo y quieres contribuir con su difusión o una aportación, quedo muy agradecido** <https://ko-fi.com/javierfpanadero>.

Otros programas de ejemplo

Master Mind

Es un juego en el que se elige una combinación al azar de cuatro elementos entre un conjunto conocido (números o colores) que el oponente debe adivinar. Para darle pistas en los sucesivos intentos, se le dirá si cada elemento que puso estaba en la combinación elegida y en la posición que lo sitió, o bien estaba en la combinación elegida aunque en otra posición, o bien no estaba en la combinación elegida.

```
'''En esta versión del Master Mind hay que adivinar una
combinación de cuatro números elegidos al azar.
Para cada prueba te dirá si cada número elegido:
- No está en la combinación (X)
- Está en la combinación pero no en esa posición (C)
- No está en la combinación (B)
Por Javier Fernández Panadero 27/10/2018 @javierfpanadero'''
```

```
import random

def elegirCombinacion():
    posibles = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
    combinacion = random.sample(posibles, 4) #toma cuatro
    elementos desordenados de la lista posibles
    return combinacion
```

(SIGUE EN LA PÁGINA SIGUIENTE)

```
def comprobarCombinacion(test):
    cuenta = []
    for i in range(4): #para cada número del test
        for j in range(4): #para cada número de la clave
            if (test[i] == clave[j]): #si son iguales
                if (i == j): #y sus índices también
                    cuenta.append('B') #valor y posición
correctos
                else:
                    cuenta.append('C') #si no coinciden los
índices, sólo valor correcto
            if (len(cuenta) <= i): #si llego al final del último
valor de j y no he añadido una B o una C a la lista cuenta, es
que el número en la posición i no está así que añado una X
                cuenta.append('X')
    return cuenta

def capturarRespuesta():
    respuesta = input('Intenta adivinar la combinación, separada
con espacios ')
    respuesta_separada = [int(x) for x in respuesta.split()]
#separo la cadena en enteros
    return respuesta_separada

print(';Juguemos al Master Mind!\nElegiré una combinación de
cuatro números entre 0 y 9. Sin repetir')
clave = elegirCombinacion()
print(clave) #por ver que funciona bien
prueba = []
while (prueba != clave): #compruebo hasta que acierte
    prueba = capturarRespuesta()
    resultado = comprobarCombinacion(prueba)
    print(prueba)
    print(resultado)
print('Acertaste')
```

Cómo usar Python para tratar datos en SPSS

Aquí os dejo en enlace a un artículo donde se explica una manera sencilla de extraer datos de un archivo en SPSS, hacer operaciones con ellos en Python, y cargar los resultados en otro campo del archivo SPSS.

<https://lacienciaparatodos.wordpress.com/2019/09/05/spss-y-python-como-leer-y-escribir-datos/>

Crear exámenes tipo test en Moodle o papel con datos variables

Con este programa podéis escribir de forma sencilla un examen en texto plano y el programa te generará tantas versiones diferentes como quieras en papel, cambiando datos incluso, o un archivo XML para subirlo a un entorno Moodle.

<https://lacienciaparatodos.wordpress.com/2022/07/02/programa-generador-de-examenes-en-papel-y-en-moodle/>

Para ampliar

AUTOMATE THE BORING STUFF WITH PYTHON

Personalmente soy muy fan del libro Automate the boring stuff with Python por el enfoque de aplicabilidad en tareas cotidianas más allá de los programas ad hoc o de la actividad específica del programador profesional.

Empieza desde cero, con lo cual puede ser abordado por cualquiera.

<https://www.w3schools.com/python/default.asp>

Curso online interactivo de bastante alcance.

<http://docs.python.org.ar/tutorial/3/index.html>

Manual muy completo.

Javier Fernández Panadero
lacienciaparatodos.wordpress.com
[@javierfpanadero](https://twitter.com/javierfpanadero)