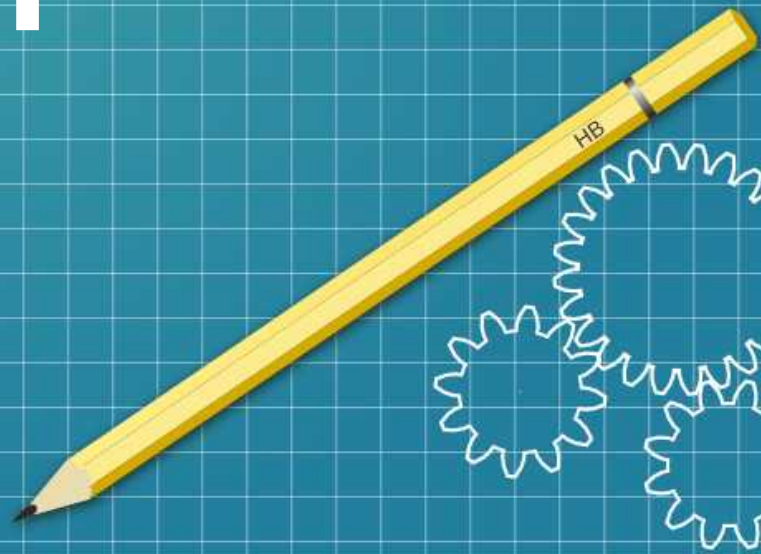


Python



Java vs Python



- Java: Lenguaje orientado a objetos
- Java es un lenguaje compilado
- Para ejecutar código hay que instalar Java Virtual Machine.
- Java usa tipos estáticos: Si declaras una variable como int se queda así para siempre.
- No puedes escribir código sin definir primero una clase.
- Python: Lenguaje orientado a objetos
- Python es un lenguaje dinámico interpretado
- Python usa tipos dinámicos: una misma variable puede contener un número, luego una cadena de texto y finalmente una lista.
- Se puede escribir código fuera de una clase o función. Puedes poner un print nada más empezar.

Java vs Python



- Aprendizaje de Java: Muchas estructuras que hay que conocer de forma detallada. Se tarda más en aprender.
- Rendimiento de Java: Es más rápido de ejecutar. Se compila y ejecuta en MV de Java.
Ejemplo: Netflix usa Java para ejecutar el homecinema.

- Aprendizaje de Python: Menos estricto, más flexible e intuitivo. Se tarda menos en aprender.
- Rendimiento de Python: Es más lento de ejecutar. Se interpreta, se compila y finalmente se ejecuta.
Ejemplo: Netflix usa Python para resolver problemas de la red, el control y manejo del contenido, control y administración de usuarios, etc.

Java vs Python



- Sintaxis Java: más detallada y estricta, hay que definirlo todo. Necesita más cantidad de código.

```
public class Test {  
    public static void main(String[] args) {  
        // Obtencion de valores  
        Scanner scr = new Scanner(System.in);  
        System.out.println("Dame un valor: ");  
        int a = Integer.parseInt(scr.nextLine());  
        System.out.println("Dame otro valor: ");  
        int b = Integer.parseInt(scr.nextLine());  
        // Recorrido y sumatoria  
        int sumatorio = 0;  
        for (int num = a; num < b; num++) {  
            sumatorio += num;  
            System.out.println(num);  
        }  
        System.out.println(String.format("El sumatorio es %d", sumatorio));  
    }  
}
```

- Sintaxis Python: más simple y flexible.
Necesita menos cantidad de código.

```
def main():  
    a = int(input("Introduce un valor: "))  
    b = int(input("Introduce otro valor: "))  
    for x in range(a,b):  
        print(x)  
    print(f'El sumatorio es {sum(range(a,b))}')  
  
if __name__ == '__main__':  
    main()
```

Elementos de un programa de Python



- Líneas y espacios
- Un programa de Python está formado por líneas de texto.
- Se recomienda que cada línea contenga una única instrucción.
- Los elementos del lenguaje se separan por espacios en blanco (normalmente, uno), aunque en algunos casos no se escriben espacios:
 - entre los nombres de las funciones y el paréntesis
 - antes de una coma (,)
 - entre los delimitadores y su contenido (paréntesis, llaves, corchetes o comillas)

```
radio = 5  
area = 3.14159242 * radio ** 2  
print(area)
```


Elementos de un programa de Python



- Excepto al principio de una línea, los espacios no son significativos.
- Los espacios al principio de una línea (el sangrado) son significativos porque indican un nivel de agrupamiento.
- Otros lenguajes utilizan un carácter para delimitar agrupamientos, en muchos se utilizan las llaves { }
- Una línea no puede contener espacios iniciales, a menos que forme parte de un bloque de instrucciones o de una instrucción dividida en varias líneas.

Elementos de un programa de Python



- Ejemplo de agrupamientos con tabulador:

```
while True:
    try:
        numero = int(input("Dígame un número entero: "))
        break
    except ValueError:
        print("No ha escrito un número entero")
print(numero)
```

- Para más información:
 - <https://mclibre.org/consultar/python/lecciones/python-elementos.html>

Programa básico de Python



- Las instrucciones que forman el programa, se escribirían dentro de la función main().

```
def main():  
    print("¡Hola, mundo!")  
    print("¡Adiós, mundo!")  
  
if __name__ == "__main__":  
    main()
```

- Más información en:
 - <https://mclibre.org/consultar/python/lecciones/python-plantilla.html>

Salida por pantalla: la función print()



- La función print() permite mostrar texto en pantalla.

```
print("Hola")
```

- Las cadenas se pueden delimitar tanto por comillas dobles (") como por comillas simples (').

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

- La función print() admite varios argumentos seguidos. En el programa, los argumentos deben separarse por comas.

```
print("Hola", "Adiós")
```

Salida por pantalla: la función print()



- Al final de cada print(), Python añade automáticamente un salto de línea. (argumento end="salto de línea" por defecto):

```
print("Hola")
```

```
print("Adiós")
```

```
Hola
```

```
Adiós
```

- Si se quiere que Python en print() añada al final otra cosa, hay que darle valores al argumento end, por ejemplo "no espacio" end="":

```
print("Hola. ", end="")
```

```
print("Adiós")
```

```
HolaAdios
```

end=" " (un espacio)

```
print("Hola. ", end=" ")
```

```
print("Adiós")
```

```
Hola. Adiós
```

Salida por pantalla: la función print()



- f da formato a la cadena que está a continuación y para añadir variables a la cadena hay que hacerlo con { }
- Ejemplo de print con 3 argumentos (cadena,variable,cadena):

```
nombre = "Pepe"
```

```
print("¡Hola,", nombre, "!")
```

```
¡Hola, Pepe!
```

La función print() muestra los argumentos separados por espacios, lo que a veces no es conveniente.

- Ejemplo de print() con 1 argumento (1 cadena):

```
nombre = "Pepe"
```

```
print(f"¡Hola, {nombre}!")
```

```
¡Hola, Pepe!
```

Entrada por teclado: la función input()



- La función input() permite obtener texto escrito por teclado.
- El usuario escribe su respuesta en una línea distinta a la pregunta porque Python añade un salto de línea al final de cada print().

```
print("¿Cómo se llama?")  
nombre = input()  
print(f"Me alegro de conocerle, {nombre}")
```

¿Cómo se llama?

Pepe

Me alegro de conocerle, Pepe

Entrada por teclado: la función input()



- Si se prefiere que el usuario escriba su respuesta a continuación de la pregunta, se podría utilizar el argumento opcional end en la función print()

```
print("¿Cómo se llama? ", end="")  
nombre = input()  
print(f"Me alegro de conocerle, {nombre}")
```

```
¿Cómo se llama? Pepe  
Me alegro de conocerle, Pepe
```

- Otra solución, más compacta, es aprovechar que a la función input() se le puede enviar un argumento

```
nombre = input("¿Cómo se llama? ")  
print(f"Me alegro de conocerle, {nombre}")
```

```
¿Cómo se llama? Pepe  
Me alegro de conocerle, Pepe
```

Función input() - Conversión de tipos



- De forma predeterminada, la función input() convierte la entrada en una cadena, aunque escribamos un número.
- Si se quiere que Python interprete la entrada como un número entero, se debe utilizar la función int() de la siguiente manera:

```
cantidad = int(input("Dígame una cantidad en pesetas: "))  
print(f"{cantidad} pesetas son {round(cantidad / 166.386, 2)} euros")
```

```
Dígame una cantidad en pesetas: 500  
500 pesetas son 3.01 euros
```


Función input() - Conversión de tipos



- De la misma manera, para que Python interprete la entrada como un número decimal, se debe utilizar la función float() de la siguiente manera:

```
cantidad = float(input("Dígame una cantidad en euros (hasta con 2 decimales): "))  
print(f"{cantidad} euros son {round(cantidad * 166.386)} pesetas")
```

```
Dígame una cantidad en euros (hasta con 2 decimales): 9.99  
9.99 euros son 1662 pesetas
```

Función input() - Conversión de tipos



- Pero si el usuario no escribe un número, las funciones int() o float() producirán un error:

```
cantidad = float(input("Dígame una cantidad en euros (hasta con 2 decimales): "))  
print(f"{cantidad} euros son {round(cantidad * 166.386)} pesetas")
```

Dígame una cantidad en euros (hasta con 2 decimales): **Pepito**

Traceback (most recent call last):

File "ejemplo.py", line 1, in <module>

cantidad = float(input("Dígame una cantidad en euros: "))

ValueError: could not convert string to float: 'Pepito'

Función input() - Conversión de tipos



- Si el usuario escribe un número decimal, la función int() producirá un error:

```
edad = int(input("Dígame su edad: "))  
print(f"Su edad son {edad} años")
```

Dígame su edad: 15.5

Traceback (most recent call last):

File "ejemplo.py", line 1, in <module>

edad = int(input("Dígame su edad: "))

ValueError: invalid literal for int() with base 10: '15.5'

Función input() - Conversión de tipos



- Pero si el usuario escribe un número entero, la función float() no producirá un error, aunque el número se escribirá con parte decimal (.0):

```
peso = float(input("Dígame su peso en kg: "))  
print(f"Su peso es {peso} kg")
```

```
Dígame su peso en kg: 84
```

```
Su peso es 84.0 kg
```

Bucle for (1)



- Un bucle es una estructura de control que repite un bloque de instrucciones. Un bucle for es un bucle que repite el bloque de instrucciones un número predeterminado de veces.
- En Python no existe `i++`; o `i--`;
- En Python el cuerpo del bucle se ejecuta tantas veces como elementos tenga el elemento recorrible (elementos de una lista o de un `range()`, caracteres de una cadena, etc.).

Bucle for (1)



- El bucle for se repite tantas veces como elementos tiene una lista:

```
print("Comienzo")  
for i in [0, 1, 2]:  
    print("Hola ", end="")  
print()  
print("Final")
```

```
Comienzo  
Hola Hola Hola  
Final
```


Bucle for (1)



- Los valores que toma la variable no son importantes, lo que importa es que la lista tiene tres elementos y por tanto el bucle se ejecuta tres veces. El siguiente programa produciría el mismo resultado que el anterior:

```
print("Comienzo")  
for i in [1, 1, 1]:  
    print("Hola ", end="")  
print()  
print("Final")
```

```
Comienzo  
Hola Hola Hola  
Final
```

Bucle for (1)

- Si la lista está vacía, el bucle no se ejecuta ninguna vez. Por ejemplo:

```
print("Comienzo")  
for i in []:  
    print("Hola ", end="")  
print()  
print("Final")
```

Comienzo

Final

Bucle for (1)



- Si las variables de la lista nos pueden servir para alguna instrucción, podemos usar los valores en el cuerpo del bucle. Hay que tener en cuenta que la variable de control va tomando los valores del elemento recorrible. Por ejemplo:

```
print("Comienzo")
for i in [3, 4, 5]:
    print(f"Hola. Ahora i vale {i} y su cuadrado {i ** 2}")
print("Final")
```

Comienzo

Hola. Ahora i vale 3 y su cuadrado 9
Hola. Ahora i vale 4 y su cuadrado 16
Hola. Ahora i vale 5 y su cuadrado 25
Final

Bucle for (1)



- La lista puede contener cualquier tipo de elementos, no sólo números:

```
print("Comienzo")  
for i in ["Alba", "Benito", 27]:  
    print(f"Hola. Ahora i vale {i}")  
print("Final")
```

```
Comienzo  
Hola. Ahora i vale Alba  
Hola. Ahora i vale Benito  
Hola. Ahora i vale 27  
Final
```

Bucle for (1)



- Para definir las vueltas que dará un bucle `for()`, a no ser que se quieran usar los valores en el cuerpo del bucle, se recomienda usar `range()`. Con esta opción se ocupa menos memoria interna que en los ejemplos anteriores.
- `range()` controla el número de veces que se ejecuta el bucle.
- `range(n)` va de 0 a $n-1$.
- `range(5)`: 0,1,2,3,4

Bucle for (1)

- Este bucle se ejecutará 10 veces y la i valdrá:
(0,1,2,3,4,5,6,7,8,9)

```
print("Comienzo")  
for i in range(10):  
    print("Hola ", end="")  
print()  
print("Final")
```

Comienzo

Hola Hola Hola Hola Hola Hola Hola Hola Hola

Final

Bucle for (1)



- Se puede también pedir al usuario cuántas veces quiere que se ejecute el bucle:

```
veces = int(input("¿Cuántas veces quiere que le salude? "))  
for i in range(veces):  
    print("Hola ", end="")  
print()  
print("Adiós")
```

```
¿Cuántas veces quiere que le salude? 6  
Hola Hola Hola Hola Hola Hola  
Adiós
```

Bucle for (1)



- Bucle for() para implementar:
 - **Testigo**: una variable que indica simplemente si una condición se ha cumplido o no.
 - **Contador**: una variable que lleva la cuenta del número de veces que se ha cumplido una condición.
 - **Acumulador**: una variable que acumula el resultado de una operación.
- Ver los ejemplos al final de esta página:
 - <https://mclibre.org/consultar/python/lecciones/python-for.html>

Bucle for (2) - Bucles anidados

- Se habla de bucles anidados cuando un bucle se encuentra en el bloque de instrucciones de otro bloque.
- Al bucle que se encuentra dentro del otro se le puede denominar bucle interior o bucle interno. El otro bucle sería el bucle exterior o bucle externo.
- Recuerda que aquí no hay { } que delimiten los bucles, se pasa de un bucle a otro dependiendo del número de tabuladores.

Bucle for (2) - Bucles anidados



```
for i in [0, 1, 2]:  
    for j in [0, 1]:  
        print(f"i vale {i} y j vale {j}")
```

```
i vale 0 y j vale 0  
i vale 0 y j vale 1  
i vale 1 y j vale 0  
i vale 1 y j vale 1  
i vale 2 y j vale 0  
i vale 2 y j vale 1
```

Bucle for (2) - Bucles anidados



- En Python se puede incluso utilizar la misma variable en los dos bucles anidados porque Python las trata como si fueran dos variables distintas. **Aunque mejor usar nombres de variables diferentes.**

```
for i in range(3):  
    print(f"i (externa) vale {i}")  
    for i in range(2):  
        print(f"i (interna) vale {i}", end=" ")
```

```
i (externa) vale 0  
i (interna) vale 0 i (interna) vale 1  
i (externa) vale 1  
i (interna) vale 0 i (interna) vale 1  
i (externa) vale 2  
i (interna) vale 0 i (interna) vale 1
```

Bucle for (2) - Bucles anidados

- Se dice que las variables de los bucles son dependientes cuando los valores que toma la variable de control del bucle interno dependen del valor de la variable de control del bucle externo. Por ejemplo:

```
for i in [1, 2, 3]:  
    for j in range(i):  
        print(f"i vale {i} y j vale {j}")
```

| | |
|---------------------|-----------------|
| i vale 1 y j vale 0 | ← j in range(1) |
| i vale 2 y j vale 0 | ← j in range(2) |
| i vale 2 y j vale 1 | |
| i vale 3 y j vale 0 | ← j in range(3) |
| i vale 3 y j vale 1 | |
| i vale 3 y j vale 2 | |

Bucle for (2) - Bucles anidados

- Para más ejemplos de bucles anidados:
 - <https://mclibre.org/consultar/python/lecciones/python-for-2.html>

Bucle while

- Un bucle while permite repetir la ejecución de un grupo de instrucciones mientras se cumpla una condición (es decir, mientras la condición tenga el valor True).
- La sintaxis del bucle while es la siguiente:

while condicion:
 cuerpo del bucle



Bucle while



- Para ver más ejemplos:
 - <https://mclibre.org/consultar/python/lecciones/python-while.html>

```
i = 1
while i <= 3:
    print(i)
    i += 1
print("Programa terminado")
```

```
1
2
3
Programa terminado
```

Bucle while

- Para evitar bucles infinitos que no acaban nunca:
 - Incrementar la variable que se evalúa en la condición.
 - Poner una condición que tenga un tope o un fin
($i > 0$ es un ejemplo de condición que **no acaba nunca**)

```
i = 1
while i != 12: ← condición que acaba
    print(i, end=" ")
    i += 2      ← incrementar la variable
```

1 2 3 4 5 6 7 8 9 10 11

Funciones en Python



- Las funciones en Python admiten argumentos en su llamada y permiten devolver valores.
- Cuando la función no retorna nada, se podría prescindir de escribir **return**, pero es aconsejable ponerlo.

```
def nombre_funcion(argumento1,argumento2,...):  
    Instrucciones  
    return resultado
```

Funciones en Python



- Las funciones en Python admiten argumentos en su llamada y permiten devolver valores.

```
def licencia():  
    print("Copyright 2020 Bartolomé Sintes Marco")  
    print("Licencia CC-BY-SA 4.0")  
    return  
  
print("Este programa no hace nada interesante.")  
licencia()  
print("Programa terminado.")
```

```
Este programa no hace nada interesante.  
Copyright 2020 Bartolomé Sintes Marco  
Licencia CC-BY-SA 4.0  
Programa terminado.
```

Funciones en Python



```
def calcula_media(x, y):  
    resultado = (x + y) / 2  
    return resultado
```

```
a = 3  
b = 5  
media = calcula_media(a, b)  
print(f"La media de {a} y {b} es: {media}")  
print("Programa terminado")
```

```
La media de 3 y 5 es: 4.0  
Programa terminado
```


Funciones en Python



- Las funciones pueden admitir una cantidad indeterminada de valores:

```
def calcula_media(*args):
```

```
    total = 0
```

```
    for i in args:
```

```
        total += i
```

```
    resultado = total / len(args)
```

```
    return resultado
```

```
a, b, c = 3, 5, 10
```

```
media = calcula_media(a, b, c)
```

```
print(f"La media de {a}, {b} y {c} es: {media}")
```

```
print("Programa terminado")
```

La media de 3, 5 y 10 es: 6.0

Programa terminado

Funciones en Python



- Las funciones pueden devolver varios valores simultáneamente

```
def calcula_media_desviacion(*args):  
    total = 0  
    for i in args:  
        total += i  
    media = total / len(args)  
    total = 0  
    for i in args:  
        total += (i - media) ** 2  
    desviacion = (total / len(args)) ** 0.5  
    return media, desviacion
```

```
a, b, c, d = 3, 5, 10, 12  
media, desviacion_tipica = calcula_media_desviacion(a, b, c, d)  
print(f"Datos: {a} {b} {c} {d}")  
print(f"Media: {media}")  
print(f"Desviación típica: {desviacion_tipica}")  
print("Programa terminado")
```

Datos: 3 5 10 12

Media: 7.5

Desviación típica: 3.640054944640259

Programa terminado

try



- El bloque try de Python ayuda a controlar los errores para evitar que se propagen de una función a otra o de un módulo a otro.
- Si controlamos los errores y enviamos mensaje por pantalla, ayudamos a encontrar el problema y evitamos que el programa “pete” y se quede colgado.
- Como anécdota en la nave no tripulada Ariane5 se produjo un **error de overflow**. El problema fue que se reaprovechó el código del programa de vuelo del Ariane4. En cálculo de coordenadas del Ariane4 era suficiente con un registro de **16bits**. El Ariane5, que modernizó su sistema de vuelo, necesitaba poder calcular con números que ocupaban **32bits**, al reaprovechar el código del Ariane4, se pasó ese detalle por alto (porque no estaba indicado en el código original) y encima en el código no se escribió **ningún control de errores!** Esto hizo que el **error overflow** se propagara y acabó haciendo explotar el cohete. Significó una pérdida de US\$370 millones.

<https://www.youtube.com/watch?v=5tJPXYA0Nec>

try



try:

Código a ejecutar

except <tipo de error>:

Se ejecutara si el código en try lanza un error

else:

Se ejecutara si el código en try se ejecuta sin errores

finally:

Esta parte se ejecutara siempre tato si ha habido error cómo si no ha habido error

try

- Programa sin control de errores:

```
def dividir(x,y):  
    return x/y
```

```
num=int(input("Indica un número: "))  
div=int(input("Indica el divisor: "))  
dividir(num,div)  
print("El resultado es:", res)
```

Si dividimos por 0
(div=0) nos dará un error
y el programa “petará”.



try



- Programa que controla el error de dividir por 0:

```
def dividir(x,y):  
    return x/y  
  
num=int(input("Indica un número: "))  
div=int(input("Indica el divisor: "))  
try:  
    res = dividir(num, div)  
except ZeroDivisionError:  
    print("Trataste de dividir entre cero :( ")  
else:  
    print("El resultado es:", res)  
finally:  
    print("Hemos acabado para bien o para mal.")
```

Ahora si intentan dividir por 0:

- controlamos el error y el programa no peta
- avisamos al usuario que no se puede dividir por 0.

try



- Programa que controla el error de dividir por 0 y que el usuario introduzca números y no texto:

```
def dividir(x,y):  
    return x/y  
  
try:  
    num=int(input("Indica un número: "))  
    div=int(input("Indica el divisor: "))  
    res = dividir(num, div)  
except ZeroDivisionError:  
    print("Trataste de dividir entre cero :( ")  
except ValueError:  
    print("No es un número!")  
else:  
    print("El resultado es:", res)  
finally:  
    print("Hemos acabado para bien o para mal.")
```

para controlar el error, tiene que estar dentro del try

Si el usuario intentan dividir por 0:

- controlamos el error y el programa no peta.
- avisamos al usuario que no se puede dividir por 0.

Si el usuario escribe texto en vez de un número:

- controlamos el error y el programa no peta.
- avisamos al usuario que ha de escribir números.

Zip: Tratar 2 o más vectores a la vez



- La función `zip()` puede tomar cualquier iterable como argumento. Se utiliza para devolver un objeto `zip` que también es un iterador.
- El iterador devuelto se devuelve como una tupla como una lista, un diccionario o un conjunto. En esta tupla, los primeros elementos de ambos iterables se emparejan. Los segundos elementos de ambos iterables están emparejados, y así sucesivamente.

zip con 2 listas



```
a = [1, 2]  
b = ["Uno", "Dos"]  
c = zip(a, b)
```

```
print(c)  
print(list(c))
```

← Convierte la estructura que genera zip a listas (arrays).

Imprimir el resultado de zip:

```
<zip object at  
0x000000F619997340>
```

Imprimir el resultado de zip
formateado como lista:

```
[(1, 'Uno'), (2, 'Dos')]
```

zip con n listas



```
e = "elefante"  
g = "carretera"
```

```
a = [1, 2]  
b = ["Uno", "Dos"]  
c = [0.5, 0.7]  
d = [f"Un día vi a un {e}", f"volando por la {g}"]
```

```
z = zip(a, b, c, d)
```

```
print(list(z))
```

```
[(1, 'Uno', 0.5, 'Un día vi a un elefante'),  
(2, 'Dos', 0.7, 'volando por la carretera')]
```

zip tratando los 1eros, 2os, ... elementos



```
numeros = [1, 2, 3, 4]
espanol = ["Uno", "Dos", "Tres", "Cuatro"]
ingles = ["One", "Two", "Three", "Four"]
frances = ["Un", "Deux", "Trois", "Quatre"]
aleman = ["Eins", "Zwei", "Drei", "Vier"]

for (n,e,i,f) in zip(numeros, espanol, ingles, frances, aleman):
    print("Número:", n, "Español:", e, "Inglés:", i, "Francés:", f)
```

```
for (n,e,i,f) in zip(numeros,
espanol, ingles, frances,
aleman):
```


**ValueError: too many values to
unpack (expected 4)**

zip tratando los 1eros, 2os, ... elementos



```
numeros = [1, 2, 3, 4]
espanol = ["Uno", "Dos", "Tres", "Cuatro"]
ingles = ["One", "Two", "Three", "Four"]
frances = ["Un", "Deux", "Trois", "Quatre"]
aleman = ["Eins", "Zwei", "Drei", "Vier"]
```

faltaba un argumento:
zip tiene 5 iteradores
Se han de guardar en 5 variables



```
for (n,e,i,f,a) in zip(numeros, espanol, ingles, frances, aleman):
    print("Número:", n, "Español:", e, "Inglés:", i, "Francés:", f, "Alemán:", a)
```

Número: 1 Español: Uno Inglés: One Francés: Un Alemán: Eins

Número: 2 Español: Dos Inglés: Two Francés: Deux Alemán: Zwei

Número: 3 Español: Tres Inglés: Three Francés: Trois Alemán: Drei

Número: 4 Español: Cuatro Inglés: Four Francés: Quatre Alemán: Vier

Ejercicio



- Escribe 1 lista de animales y 1 lista de colores.
- Con zip imprime una fase donde aparezcan las 2 listas combinadas.
- Extra: si usas la función split(), puedes pedir por pantalla que el usuario te pase la lista de animales y la lista de colores separados por espacios. Luego imprimir con zip la combinación de ambas listas ¿Lo quieres intentar?