INTRODUCCIÓN.

```
#!/usr/bin/env python
```

```
print("Hello, world!") # Imprime en pantalla: Hello, world!.
```

Variables.

```
message = "Hello, Python world."
print(message)
```

Cambiamos el valor de la variable. Podemos cambiar el valor de una variable en cualquier momento.

```
message = "Hello, Python Crash course world."
```

Los nombres de las variables solo pueden contener letras, números y guiones bajos. No pueden empezar por un número: "1_variable" no es válido. Las mayúsculas en los nombres de variables tienen significados especiales.

```
print(message)
```

Error en el código.

```
message = "Prueba de error"
#print(mesage) #NameError:name 'mesage' is not defined. Did you mean: 'message'?
```

Cadenas (strings).

Una cadena es una serie de caracteres. En Python, cualquier cosa entre comillas es una cadena. Se pueden usar comillas simples o dobles para delimitarlas. Esta flexibilidad nos permite usar comillas y apóstrofos dentro de las cadenas.

```
message_1 = 'Le dije a mi amigo: "Python es mi lenguaje favorito."'
message_2 = "El lenguaje 'Python' se llama así por Monty Python."
message_3 = "It's OK."
```

Impresión concatenada.

```
print(message_1 + " - " + message_2 + " - " + message_3)
```

Uso de métodos.

Un método es una acción que se puede realizar con datos. El '.' detrás de var en var.title() dice a Python que haga actuar el método title() sobre la variable var. Normalmente, los métodos suelen requerir información adicional para operar, esa información se coloca entre paréntesis. El método title() no requiere información adicional.

```
var = "isMaEl seRRANo"
print(var.title())  # Imprime: Ismael Serrano.
print(var.upper())  # Imprime: ISMAEL SERRANO.
```

El método lower() es especialmente útil para almacenar datos. No conviene fiarse del uso de mayúsculas que hagan los usuarios, así que podemos convertir la cadena en minúsculas para guardarla, más tarde podemos usar la grafía con mayor sentido para mostrar la cadena.

```
print(var.lower())  # Imprime: ismael serrano.
```

Uso de variables en strings. Concatenación de variables. STRINGS f.

```
first_name = "federico"
last_name = "garcía"
last_name_2 = "lorca"
```

Estos strings se denominan "strings f". f(formato), porque Python formatea la cadena reemplazando el nombre de las variables entre llaves con su valor.

```
full_name = f"{first_name} {last_name} {last_name_2}"
print(full_name.title())
```

Ejemplo de uso de un string f (cadena f).

Composición de mensajes completos haciendo uso de variables.

```
first_name = "ISmaEL"
last_name = "SeRRaNO"
full_name = f"{first_name} {last_name}"
print(f"Hello, {full_name.title()}!")
```

Podemos usar las cadenas f para componer un mensaje y luego asignarlo a una variable.

```
message = f"Hola, {full_name.title()}. Asignando un 'string f' a una variable."
print(message)
```

Nota: los strings f pertenecen a Python 3.6. En versiones anteriores se usa el método format() del siguiente modo:

```
full_name = "{} {}".format(first_name, last_name)
print(full_name.title())
```

f-strings.

Los literales de cadena formateados (o f-strings) te permiten incluir expresiones dentro de tus cadenas. Justo antes de la cadena; introduce un f o F para indicar al programa que debe usar f-string. Son los que acabamos de ver.

```
var = "f-string"
var2 = f"Esto es una cadena realizada con {var}."
print(var2)  # Imprime: Esto es una cadena realizada con f-string.
```

Operación de sustitución avanzada - formateo de cadenas -.

Las cadenas admiten una operación de sustitución avanzada conocida como formateo.

```
var = '%s, eggs, and %s' % ('spam', 'SPAM')
print(var)  # Imprime: spam, eggs, and SPAM
print("Hola, mi nombre es %s." % "Sonia")  # Imprime: Hola, mi nombre es Sonia.
```

%s actúa como marcador para el valor real. Solo necesitas indicar el valor real después del operador %.

Añadiendo espacios en blanco a un string con tabulaciones y nuevas líneas. Es algo muy práctico cuando hay mucha salida por pantalla.

Añadir tabulación a un string "\t".

```
print("Python")
print("\tPython")
```

Añadir nueva línea a un string "\n".

```
print("Lenguajes:\nPython\nC\nJavasscript")
print()
```

Combinación de ambos en un único string.

```
print("Lenguajes:\n\tPython\n\tC\n\tJavascript")
```

Eliminación de espacios en blanco: 'Python' y 'Python ' no son el mismo string, esto es muy significativo. Python puede detectar espacios en blanco innecesarios tanto a la izquierda como a la derecha del string.

Se una el método rstrip() para eliminar las espacios en blanco del lado derecho del string.

```
lenguage = "Python "
print({lenguage})
lenguage = lenguage.rstrip()
print({lenguage})
```

El método lstrip() elimina los espacios en blanco de la izquierda.

```
lenguage = "  Python "
print({lenguage})
lenguage = lenguage.lstrip()
print({lenguage})
```

El método strip() elimina espacios en blanco a izquierda y derecha.

```
lenguage = "  Python "
print({lenguage})
lenguage = lenguage.strip()
print({lenguage})
```

Estos métodos se usan para limpiar la entrada del usuario antes de que se almacene la información.

Evitar errores de sintaxis con strings.

```
message = "One of Python's strengths is its diverse community."
print(message)
```

Por ejemplo, si utiliza un apóstrofo entre comillas simples, producirá un error. Esto sucede porque Python interpreta todo entre la primera comilla simple y el apóstrofo como una cadena. Luego trata de interpretar el resto del texto como código de Python, que provoca errores.

Aquí se explica cómo usar comillas simples y dobles correctamente:

```
#message = 'One of Python's strengths is its diverse community.'#SyntaxError:
                                                                #invalid syntax.
#print(message)
```

Los errores de sintaxis también son el tipo menos específico de error, por lo que pueden ser difíciles y frustrantes de identificar y corregir.

Triple comilla.

Python también tiene un formato literal de cadena entre comillas triples, a veces llamado 'cadena de bloque', esta es una conveniencia sintáctica para codificar datos de texto de varias líneas. Lo que escribe es realmente lo que obtienes.

```
a = """Hola,
    mundo"""
print(a)     # Imprime: Hola,
             #       mundo.
```

Las cadenas entre comillas triples también se usan comúnmente para cadenas de documentación, que son literales de cadena que se toman como comentarios cuando aparecen en puntos específicos de su archivo (como este párrafo).

Operaciones básicas con cadenas.

```
print('abc' + 'def')     # Imprime: 'abcdef'.
print('¡No!' * 4)        # Imprime: ¡No!¡No!¡No!¡No!.
```

También es posible iterar sobre cadenas en bucles usando declaraciones de prueba para caracteres y subcadenas con el operador de expresión 'in', que es esencialmente una búsqueda de subcadenas. Por ejemplo:

```
my_job = 'hacker'
for c in my_job:          # Bucle para recorrer elementos. [*]
    print(c, end='')      # Imprime: hacker. [**]

print('k' in my_job)      # Imprime: True.
print('z' in my_job)      # Imprime: False.
```

[*] El ciclo for asigna una variable a elementos sucesivos en una secuencia (aquí, una cadena) y ejecuta una o más instrucciones para cada elemento. En efecto, la variable 'c' se convierte en un cursor.

[**] El parámetro "end='" en la función de impresión se usa para agregar cualquier cadena al final de la salida de la declaración de impresión. Por defecto, la función de impresión termina con una nueva línea. Pasar el espacio en blanco al parámetro final (end=' '), indica que el carácter final debe identificarse con un espacio en blanco y no con una nueva línea. Así mismo, (end='') indica la ausencia de carácter final.

Indexando y cortando strings.

Dado que las cadenas se definen como colecciones ordenadas de caracteres, podemos acceder a sus componentes por posición. En Python, los caracteres de una cadena se recuperan indexando: proporcionando el desplazamiento numérico del componente deseado entre corchetes después del string obtiene la cadena de un carácter en la posición especificada.

Al igual que en el lenguaje C, las compensaciones de Python comienzan en 0. Python le permite obtener elementos de secuencias como cadenas usando compensaciones negativas. Técnicamente, se agrega una compensación negativa a la longitud de una cadena para derivar un desplazamiento positivo. También puede pensar en las compensaciones negativas como contando hacia atrás desde el final. Por ejemplo, en el siguiente código, S[-2] obtiene el elemento en el desplazamiento 2 desde el final (o de manera equivalente, en desplazamiento (4 + (-2)) desde el frente).

El corte de la cadena (slice) [:] nos permite extraer una sección completa (subcadena) en un solo paso haciendo uso del operador ':', obtiene los elementos desde 0 hasta el final; así, por ejemplo, [:] se refiere a la cadena completa.

```
S = 'spam'
print(S[0], S[-2])                    # Imprime: s a.
print(S[1:3], S[1:], S[:-1], S[:])    # Imprime: pa pam spa spam.
```

Las expresiones de división tienen soporte para un tercer índice opcional, usado como un paso (a veces llamado zancada). El paso se agrega al índice de cada elemento extraído. La forma completa de un segmento ahora es X[i:j:k]. El tercer límite, k, por defecto es 1, dado que normalmente todos los elementos de un segmento se extraen de izquierda a derecha. Si especifica un valor explícito, sin embargo, puede utilizar el tercer límite para omitir elementos o para invertir su orden.

```
S = 'abcdefghijklmnop'
print(S[1:10:2])     # Imprime: 'bdfhj'.
print(S[::2])        # Imprime: 'acegikmo'.

S = 'hello'
print(S[::-1])       # Imprime: 'olleh', la cadena invertida.
```

Números.

Enteros. Operaciones +, -, *, /, ** (potenciación) y % (resto).

```
number = 2+2
print(number)
number = 3-2
print(number)
number = 4*2
print(number)
number = 4/2  # La división siempre devuelve un float, aquí, 2.0
print(number)
number = 2**3
print(number)
number = 2 % 2

# Operador módulo. Devuelve el resto.
# Se puede usar para comprobar si un número es par. Un par es divisible entre 2
# y el resto es 0.(if number % 2 == 0) es True.

print(number)
number = 2+2*3
print(number)
number = (2+2)*3  # Cambio en la precedencia de las operaciones.
print(number)
```

Números en coma flotante, floats.

¿Por qué se llaman números en coma flotante? Flotante se refiere al hecho de que un punto decimal puede aparecer en cualquier posición del número.

```
number = 2+2.0
# Si en una operación hay un float, se devuelve un float.
# Incluso si la salida es un entero.
print(number)   # Imprime 4.0.
```

Uso del guión bajo '_' para marcar separador de miles.

```
universe_age = 14_000_000_000  # Ojo, para Python 1_000 es lo mismo que 10_00.
print(universe_age)  # Los guiones bajos son ignorados en la salida.
```

Asignación múltiple de variables.

```
x, y, z = 1, 2, 3   # Siempre que el número de valores coincida con el número de
                    # variables, Python emparejará correctamente.
print(f"{x} + {y} + {z}")
```

Además, Python ha desarrollado algunos tipos numéricos nuevos: números decimales (números de coma flotante de precisión fija) y números fraccionarios (números racionales con numerador y denominador). Ambos pueden usarse para sortear las limitaciones e imprecisiones inherentes a la coma flotante.

Notación hexadecimal, octal y binaria.

```
hex = 0xFF          # Representación hexadecimal de 255.
```

```
oct = 0o20          # Representación octal de 16.
bin = 0b010101      # Representación binaria de 21.
print(f"{hex} - {oct} - {bin}")  # Imprime: 255 - 16 - 21.
```

NumPy.

NumPy es una extensión opcional para Python (Numeric Python) proporciona herramientas de programación numérica avanzada, como un tipo de datos de matriz, procesamiento de vectores y computación sofisticada.

Constantes.

Una constante es una variable cuyo valor se mantiene durante todo el programa. El tipo constante no existe en Python, pero la norma es nombrar a las constantes en mayúsculas.

```
MAX_CONNECTIONS = 5_000
print(MAX_CONNECTIONS)
```

Ejercicio:

Usar una variable para almacenar un número. Crear un mensaje usando la variable que muestre el número e imprimirlo.

```
number = 5
# message = "El número es: {number}".
message = f"El número indicado es: {number}"
# Esto no funcionaría.
print(message)
```

LISTAS

Como hemos aprendido, las cadenas son secuencias inmutables: no se pueden cambiar, y son colecciones ordenadas posicionalmente a las que se accede por desplazamiento. Ahora bien, sucede que todas las secuencias que estudiaremos en adelante, responden a las mismas operaciones de secuencia que hemos visto para las cadenas: concatenación, indexación, iteración, etc. Más formalmente, hay tres categorías principales de tipos (y operaciones) en Python:

1. Números: que soportan adición, multiplicación, etc.

2. Secuencias (strings, listas y tuplas): que soportan indexación, cortes, concatenación, etc.

3. Asignaciones (diccionarios): que soportan indexación por clave, etc.

En pocas palabras, estas operaciones funcionan de la misma manera en cualquier tipo, incluyendo cadenas, listas, tuplas y algunos tipos de objetos definidos por el usuario. La única diferencia es que el nuevo objeto de resultado que se obtiene es del mismo tipo que los operandos, al concatenar listas, se obtiene una nueva lista, no una cadena. Indexación, corte y otras operaciones también funcionan de la misma manera en todas los tipos; el tipo de los objetos que son procesados le dice a Python qué tarea debe realizar.

Listas [].

Las listas le permiten almacenar conjuntos de información en un solo lugar, ya sea que tenga unos pocos elementos o millones de artículos. Las listas son una de las características más poderosas de Python, fácilmente accesibles a los nuevos programadores, y unen muchos e importantes conceptos en programación. Una lista es una colección de elementos en un orden particular. En Python, los corchetes ([]) indican una lista y los elementos de la lista están separados por comas.

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles) # Imprime: ['trek', 'cannondale', 'redline', 'specialized'].
```

Pero no es este el resultado que se espera en la salida, incluidos los ([]). Para solucionarlo podemos acceder a elementos individuales de la lista.

Accediendo a elementos de una lista.

```
bicycles = ['trek', 'cannondale', 'redline', 'specialized']
print(bicycles[0]) # Imprime: trek. Ahora sí, es más legible, un resultado
                   # formateado y limpio.
print(bicycles[0].title())
```

Python tiene una sintaxis especial para acceder al último elemento de una lista. Preguntando por el elemento de índice -1, siempre devuelve el último elemento de la lista. Esto es especialmente útil cuando no se conoce el número de elementos de una lista y se quiere acceder al último, penúltimo, ...

```
print(bicycles[-1])
print(bicycles[-2])          # Devuleve: "redline".
print(bicycles[-3].title()) # Devuelve: "Cannondale".
```

Ejemplos de uso:

```
message = f"Mi primera bicicleta fue una '{bicycles[1].title()}'"
print(message) # Devuelve: Mi primera bicicleta fue una 'Cannondale'
```

Modificar, añadir y eliminar elementos de una lista.

La mayoría de las listas que cree serán dinámicas, lo que significa que creará una lista y luego agregará y eliminará elementos de ella a medida que el programa siga su curso.

Modificar un elemento.

```
persons = ['Juan', 'Manuel', 'Rodrigo', 'Víctor']
print(persons[1])
persons[1] = "Jorge"
print(persons)
```

Añadir un elemento al final de la lista. Método append().

```
persons.append('Alberto')
print(persons[-1])
print(persons)
```

Creación dinámica de listas haciendo uso del método append().

```
motorcycles = []
print(motorcycles)
motorcycles.append("Honda") # Notar que es indistinto el uso de comillas simples
                            # ('') o comillas dobles ("").
motorcycles.append('Ducati')
motorcycles.append('Vespa')
print(motorcycles)
```

Añadir un elemento en una posición determinada. Método insert().

```
motorcycles.insert(1, 'Suzuki') # Añadimos 'Suzuki' como elemento [1] de la
lista.
print (motorcycles)
```

Eliminación de elementos de la lista.

Es posible eliminar un elemento según su posición en la lista o según su valor.

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles)
del motorcycles[0] # Eliminación del primer elemento.
print(motorcycles)
```

Eliminar un elemento de la lista y poder seguir usándolo, método pop().
El método pop() elimina el último elemento de una lista, pero permite trabajar con ese elemento después de quitarlo. El término pop proviene de pensar en una lista como una pila de elementos y sacar un elemento de la parte superior de la pila. En esta analogía, la parte superior de una pila corresponde al final de una lista.

```
motorcycles = ['honda', 'yamaha', 'suzuki']
print(motorcycles)
popped_motorcycle = motorcycles.pop() # Removemos último elemento de la lista.
```

El resultado muestra que el valor 'suzuki' se eliminó del final de la lista y ahora está asignado a la variable popped_motorcycle.

```
print(motorcycles)
print(popped_motorcycle.title())
```

Uso práctico del método pop():

```
edad = [20, 30, 40, 46, 47]
edad_actual = edad.pop()
print(f"Actualmente mi edad es de {edad_actual} años.")
```

Uso de pop para posiciones concretas en la lista.

```
edad_anterior = edad.pop(3)
print(f"El año pasado mi edad era de {edad_anterior} años.")
```

IMPORTANTE: Recuerda que cada vez que usas pop() , el elemento con el que trabajas ya no se almacena en la lista.

```
print(edad) # Imprime [20, 30, 40].
```

Eliminando un elemento por su valor.

Existirán ocasiones en las que desconozcamos la posición de un elemento en la lista.

```
#edad.remove('47') # Error: ValueError: list.remove(x): x not in list.
#edad.remove('40') # Error: ValueError: list.remove(x): x not in list.
edad.remove(40)     # Ok.
print(edad)
```

El método remove() elimina solo la primera aparición del valor que especifica. Si existe la posibilidad de que el valor aparezca más de una vez en la lista, necesita usar un bucle para asegurarse de que se eliminen todas las apariciones del valor.

Ordenar una lista.

Ordenar una lista de forma permanente con el método sort().

sort() es un método del tipo Lista.

```
list = ['Juan', 'Pedro', 3, 'Jorge']
# list.sort() # Error:TypeError:'<'not supported between instances of 'int' and
              # 'str'. No se puede ordenador una mezcla de strings y enteros.
print(list)

parse = str(list.pop(2)) # Convertimos en string el entero con el método str(),
                         # a su vez eliminamos de la lista el entero
list.insert(3, parse) # Insertamos la cadena '3' en la lista en cuarta posición.
print(parse)
print(list) # Devuelve: ['Juan', 'Pedro', 'Jorge', '3']

list.sort() # Ordenamos la lista. Ya no se podrá revertir al orden original.
print(list) # Devuelve: ['3', 'Jorge', 'Juan', 'Pedro']
list.sort(reverse=True) # Orden inverso. Y nuevamente definitivo.
print(list) # Devulve: ['Pedro', 'Juan', 'Jorge', '3']
```

Ordenar una lista de forma temporal con el método sorted().

Para preservar el orden actual de una lista, pero mostrar la lista ordenada usamos la función sorted(). sorted() es una función de la librería estándar.

```
cars = ['bmw', 'audi', 'toyota', 'subaru']

print("Este es el orden original de la lista:")
#print({cars.title()})  # Error: AttributeError: 'list' object has no attribute
                        # 'title'. No se puede usar title() sobre la totalidad
                        # de la lista, solo sobre cada elemento.
#print(cars[0].title()) # Ok
print(cars)
print("\nAquí se presenta la lista ordenada:")
print(sorted(cars))
print("\nAquí se presenta el orden original nuevamente:")
print(cars)
```

El método sorted() también acepta el argumento 'reverse=True' si se quiere mostrar en pantalla un orden alfabético inverso.

```
print(sorted(cars, reverse=True))#Devuelve: ['toyota', 'subaru', 'bmw', 'audi'].
```

Imprimir una lista en orden inverso.

```
cars = ['bmw', 'audi', 'toyota', 'subaru']
print(cars)
cars.reverse() # No ordena alfabéticamente, solo invierte el orden.
print(cars)
```

El método reverse() cambia el orden permanentemente, pero se puede revertir usando nuevamente el mismo método reverse().

Medir la longitud de una lista.

len() es una función de la librería estándar., integrada en Python. Esto es importante, pues opera como función, no se puede usar el método de llamada a métodos: '.len()', si no que su uso es del siguiente modo: 'len(argumento)'. len() Es una función no un método.

```
print(len(cars))
print(cars.len()) #Error:AttributeError: 'list' object has no attribute 'len'.
```

TRABAJANDO CON LISTAS Y TUPLAS.

Veremos como recorrer una lista con un bucle. Cuando se desea realizar la misma acción con todos os elementos de una lista, se puede usar un bucle for.

EJERCICIO PREVIO: Cambiar los elementos de la lista a capitalización.

```
names = ["alberto", "carlos", "enrique", "luis"]
#list.title() # Error: AttributeError: 'list' object has no attribute 'title'.
```

No es un método válido, el método title() es del tipo (para) string, no del tipo list.

```
print(names)

for index, element in enumerate(names):
    names.pop(index) # Eliminamos [conservando su acceso, pop()] el elemento
                     # correspondiente al índice del bucle.
    names.insert(index, element.title()) # Insertamos capitalizado el elemento
                                         # en su posición correspondiente, según
                                         # índice del bucle, en la lista.
print (names) # Imprimimos la lista.
```

El bucle for.

```
names = ["alberto", "carlos", "enrique", "luis"]

for name in names: # Sacar un nombre de la lista y asociarlo a la variable
                   # 'name'. Repetir la operación para cada nombre de la lista.
                   # (:) Inicio de un bucle.
#print (name)# Error: IndentationError: expected an indented block after 'for'
             # statement on line 25. Se necesita identación para separar bloques
             # de código.
    print (name)
```

El conjunto de pasos se repite una vez para cada elemento de la lista, independientemente de cuántos haya. Se puede elegir cualquier nombre para la variable temporal 'name' que se asociará con cada valor de la lista. Usar nombres en singular 'name' y plural 'names' es buena idea.

Realicemos ahora una acción para cada elemento de la lista:

```
for name in names:
    print(f"Este nombre de la lista es: {name.title()}")
```

Podemos escribir todas las líneas de código que queramos dentro del bucle for. Todas las líneas sangradas bajo 'for name in names' se consideran dentro del bucle.

Nota:

La PEP8 (Python Enhancement Propusal, Propuesta de mejora de Python) recomienda usar cuatro espacios por nivel de sangrado. El intérprete de Python se confunde cuando se mezclan tabulaciones y espacios. Todos los editores de texto ofrecen usar alguna configuración que permite usar el tabulador, luego se convierte esa

tabulación en números de espacio. Mezclar tabulaciones y espacios en un archivo puede causar problemas difíciles de diagnosticar.

```
for name in names:
    print(f"Este nombre de la lista es: {name.title()}")
    print(f"{name.title()}, encantado.\n")
print('Bienvenidos los tres')
```

Cuando termina el sangrado termina el bucle.

Listas numéricas.

Función range().

```
for value in range(1, 5):
    print(value) # Observar que el 5 no es impreso. Se imprime del 1 al 4.
```

La función range() hace que Python empiece a contar en el primer valor y se detiene cuando llega al segundo valor proporcionado. Al parar en el segundo valor, la salida nunca contiene ese valor final.

```
for value in range(6):

    print(value) # Imprime del 0 al 5.
```

Imprimamos una lista:

```
numbers_1 = []
for value in range(11):
    numbers_1.append(value)
print(numbers_1) # Imprime: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10].
```

Imprimamos una lista haciendo uso de la función list():

```
numbers = list(range(0, 6))
print(numbers) # Imprime: [0, 1, 2, 3, 4, 5].
```

Elaborando estos apuntes habíamos usado 'list' como nombre de una variable. Al haber hecho eso no nos permitía hacer uso de la función list(), devolviendo el siguiente error: Error: TypeError: 'list' object is not callable.

Tamaño de paso en range()

```
even_numbers = list(range(0, 11, 2))
print(even_numbers) # Imprime los pares: [0, 2, 4, 6, 8, 10].
```

Si pasamos un tercer argumento a range(), Python toma ese valor como tamaño de paso al generar números. El siguiente ejemplo es ilustrador. Lista con el cuadrado de los números de un rango:

```
squares = []
for value in range(1, 11):
    squares.append(value ** 2)
print(squares) # Imprime: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100].
```

Algunas funciones estadísticas con listas.

```
numbers = range(100)
print(min(numbers)) # Devuelve el mínimo de la lista: 0.
print(max(numbers)) # Devuelve el máximo de la lista: 99.
print(sum(numbers)) # Devuelve la suma de la lista: 4950.
```

Listas por compresión.

Creación de listas con una única línea de código:

```
squares = [(value ** 2) for value in range(1, 11)]
squares = [value ** 2 for value in range(1, 11)] # Línea idéntica a la anterior.
print(squares) # Devuelve una lista con los cuadrados del rango.
```

Ejercicio: Imprimir de 1 a 1 millón. Sumar 1 millón.

```
numbers = range(1, 1_000_001)
for number in numbers:
    print(number)
print(sum(numbers))
```

Ejercicio: Lista de números impares entre 1 y 20. Imprimir cada número.

```
numbers = list(range(1, 21, 2))
print(numbers)
for number in numbers:
    print(number)
```

Ejercicio: Lista de los diez primeros cubos. Imprimir cada valor por separado.

```
numbers = [value**3 for value in range(1, 11)]
print(numbers)
for number in numbers:
    print(number)
```

Más sobre listas por compresión.

Además de los métodos para el tipo listas, Python incluye una operación más avanzada conocida como 'expresión de comprensión de la lista', ya hemos hablado de ella anteriormente. Resulta ser una manera poderosa de procesar estructuras como la matriz 'M' definida anteriormente.

Las listas de compresión se codifican entre corchetes [], para avisarle del hecho de que se trata de una lista y se componen de una expresión y un bucle que comparten un nombre de variable.

```
M = [[1, 2, 3],
     [4, 5, 6],
     [7, 8, 9]]

col2 = [row[1] for row in M]  # Extraemos la segunda columna.
print(col2)    # Imprime: [2, 5, 8], la segunda columna.

col2 = [row[1] for row in M if row[1] % 2 == 0] #Un filtro para elementos pares.
print(col2)    # Imprime: [2, 8].
```

La sintaxis de las lista de comprensión también puede ser usada para crear conjuntos y diccionarios, veamos unos ejemplos:

```
set = {sum(row) for row in M} # Crea un conjunto a partir de la matriz M.
dictionarie = {i : sum(M[i]) for i in range(3)}   # Crea un diccionario.

print(set)          # Imprime: {24, 6, 15}, un conjunto.
print(dictionarie)  # Imprime: {0: 6, 1: 15, 2: 24}, un diccionario.
```

Trabajar con parte de una lista. Slices.

```
players = ['alberto', 'juan', 'andres', 'jaime', 'gonzalo']
print(players[0:3]) # Imprime: ['alberto', 'juan', 'andres'].
print(players[1:4]) # Imprime: ['juan', 'andres', 'jaime'].
```

Al igual que con la función range(), Python se detiene un elemento antes del segundo índice especificado.

```
print(players[:4]) # Imprime: ['alberto', 'juan', 'andres', 'jaime'].
```

Si omitimos el primer índice, Python comienza desde el principio de la lista.

```
print(players[2:]) # Todos los elementos entre el tercero y el último.
print(players[-3:]) # Imprime los tres últimos elementos de la lista.
print(players[0:5:2]) # Imprime la posiciones impares.
print(players[0::2]) # Imprime la posiciones impares hasta el final de la lista.
```

Podemos incluir un tercer valor, como hemos hecho. Si se hace estaremos diciendo a Python cuántas veces tiene que saltar entre elementos dentro del rango.

Pasar un bucle dentro de un rango:

```
for player in players[0::2]:
    print(player.title()) # Imprime las posiciones impares dentro del rango.
```

Copiar una lista [:].

```
my_foods = ['pizza', 'falafel', 'cake']
friend_foods = my_foods[:] # Nueva lista copiando un slice de la lista anterior.
print(my_foods) # Devuleve: ['pizza', 'falafel', 'cake'].
print(friend_foods) # Devuelve: ['pizza', 'falafel', 'cake'].
my_foods.append('cannoli')
```

```
friend_foods.append('ice cream')
print(my_foods) # Devuleve: ['pizza', 'falafel', 'cake', 'cannoli'].
print(friend_foods) # Devuelve: ['pizza', 'falafel', 'cake', 'ice cream'].
```

Tenemos que usar necesariamente el slice [:]

```
my_foods = ['pizza', 'falafel', 'cake']
friend_foods = my_foods # No funciona.
print(my_foods) # Devuleve: ['pizza', 'falafel', 'cake'].
print(friend_foods) # Devuelve: ['pizza', 'falafel', 'cake'].
my_foods.append('cannoli')
friend_foods.append('ice cream')
print(my_foods) #Devuleve: ['pizza', 'falafel', 'cake', 'cannoli', 'ice cream'].
print(friend_foods)#Devuelve:['pizza','falafel','cake', 'cannoli', 'ice cream'].
```

En este caso hemos indicado que se asocie la nueva variable 'friend_foods' con la lista que ya está establecida en 'my_foods', así que ahora las dos variables apuntan a la misma lista. De ese modo la acción append se efectuará sobre ambas listas.

Tuplas ().

IMPORTANTE: Las tuplas son secuencias de elementos similares a las listas, la diferencia principal es que las tuplas no pueden ser modificadas directamente, es decir, una tupla no dispone de los métodos como append o insert que modifican los elementos de una lista. Son inmutables.

Python se refiere a los valores que no pueden cambiarse como inmutables. Una 'Lista' inmutable es una 'Tupla'.

Una lista parece una tupla, pero con paréntesis. Por ejemplo si tenemos un rectángulo que siempre debería tener el mismo tamaño, podemos asegurarnos de que no cambiará definiendo una tupla.

```
rectangle = (200, 50)
print(rectangle[0])
print(rectangle[1])
#rectangle[0] = 100 # Error: TypeError: 'tuple' object does not support item
                    # assignment. No podemos cambiar su valor.
```

Las tuplas se definen técnicamente por la presencia de una coma; los paréntesis las hacen más claras y legibles. Si queremos definir una tupla con un solo elemento, tendremos que incluir una coma.

```
rectangle = 200, 50
print(rectangle[0]) # Devuelve: 200.
print(rectangle[1]) # Devuelve: 50.

my_tuple = (3,)
print(my_tuple[0])  # Devuelve: 3.
#print(my_tuple[1]) # Error: IndexError: tuple index out of range.
```

Aunque no podemos modificar una tupla, podemos sobrescribirla. Reasignando valor a su variable. Esto sí es un método válido.

```
rectangle = (200, 50)
print(rectangle[0]) # Devuelve: 200.
print(rectangle[1]) # Devuelve: 50.

rectangle = (100, 25)
print(rectangle[0]) # Devuelve: 100.
print(rectangle[1]) # Devuelve: 25.
```

En comparación con las listas, las tuplas son estructuras de datos simples.

SENTENCIAS IF. EXPRESIONES Y PRUEBAS CONDICIONALES.

Aunque las declaraciones normalmente aparecen una por línea, es posible exprimir más de una declaración en una sola línea en Python, separándolas con punto y coma. Aunque esto solo funciona con declaraciones simples.

```
a = 1; b= 2; print(a + b)    # Tres declaraciones en una única línea.
```

Sentencias if.

Veremos como escribir pruebas condicionales, que nos permitirán comprobar cualquier condición de interés.

```
cars = ['toyota', 'seat', 'bmw', 'renault']
for car in cars:
    if car == 'bmw':#== Devuelve 'True' si los valores coinciden, 'False' si no.
                    # Un signo de igualdad simple '=' es una sentencia. Un signo
                    # de igual doble '==', en realidad, hace una pregunta.
    #if (car == 'bmw'): # Funciona igual.
    #if [car == 'bmw']: # No funcionaría.
        print(car.upper())
    else:
        print(car.title())
```

Se usan los valores 'True' y 'False' para decidir si debería ejecutarse o no el código de una sentencia if. Si se evalúa como 'True' se ejecutará el código. Con frecuencia necesitamos hacer una acción cuando se pasa una prueba condicional y otra acción en todos los demás casos 'if-else'.

Python distingue entre mayúsculas y minúsculas al comprobar una igualdad. Los sitios web usan reglas de comprobación de mayúsculas/minúsculas para los datos que introducen los usuarios, para asegurarse de que cada usuario tenga realmente un nombre de usuario único. Cuando alguien envía un nombre, se convierte a minúsculas y se compara con la versión en minúsculas de todos los nombres de usuario existentes. Esta comprobación se rechazaría para 'Juan' si ya existiese 'juan'.

Al igual que '==' comprueba una igualdad, '!=' comprueba una desigualdad.

```
for car in cars:
    if car != 'bmw':
        print(car.upper())
    else:
        print(car.title())
```

Otras comprobaciones matemáticas: <, <=, >, >=.

Operadores lógicos: 'and' y 'or'. Con ellos comprobamos varias condiciones.

Ejemplo:

```
cars = ['toyota', 'seat', 'bmw', 'renault', 'mg']
for car in cars:
    if car == 'bmw'or car == 'mg':
        print(car.upper())
    else:
        print(car.title())
```

Ejemplo:

```
age_0 = 22
age_1 = 18
print((age_0 >= 21) and (age_1 >= 21)) # Devuelve: False.
age_1 = 22
print((age_0 >= 21) and (age_1 >= 21)) # Devuelve: True.
```

Es posible que necesitemos comprobar si ya existe un nombre de usuario en una lista de usuarios antes de completar el registro del mismo. O puede ser importante saber si un valor no aparece en una lista.

Comprobar si existe un valor en una lista 'in'.

```
cars = ['toyota', 'seat', 'bmw', 'renault', 'mg']
if 'bmw' in cars:
    print('True') # Devuelve: True.
else:
    print('False') # No se ejecuta.
```

Comprobar si no existe un valor en una lista 'not in'.

```
cars = ['toyota', 'seat', 'bmw', 'renault', 'mg']
if 'skoda' not in cars:
    print('True') # Devuelve: True.
else:
    print('False') # No se ejecuta.
```

Ejemplo:

```
cars = ['toyota', 'seat', 'bmw', 'renault', 'mg']
if 'skoda' in cars:
    print('True') # No se ejecuta.
else:
    print('False') # Devuelve: False.
```

Expresiones booleanas.

Una expresión booleana es otro nombre para una prueba condicional. Un valor booleano puede ser True o False, igual que el valor de una expresión condicional. Los valores booleanos suelen usarse para seguir la pista a ciertas condiciones.

Los valores booleanos ofrecen una forma eficiente de rastrear el estado de un programa o una condición que es importante para el programa.

Ejemplo:

```
game_active = True
can_edit = False
```

if-elif-else (si- si no, si - si no)

Con frecuencia, necesitará probar más de dos situaciones posibles y para evaluarlas usamos la sintaxis de Python 'if-elif-else.

Utilizar múltiples bloques elif.

```
age = 12

if age < 4:
    price = 0
elif age < 18:
    price = 25
elif age < 65:
    price = 40
else:
    price = 20

print(f"El precio de admisión es: {price}")
```

Omitir el bloque 'else'.

Python no requiere un bloque 'else' al final de una cadena 'if-elif'. A veces, este bloque es útil; otras veces es más claro usar una sentencia 'elif' adicional que capture la condición de interés específica.

```
age = 66

if age < 4:
    price = 0
elif age < 18:
    price = 25
elif age < 65:
    price = 40
elif age >= 65:
    price = 20

print(f"El precio de admisión es: {price}")
```

Probar múltiples condiciones.

La cadena if-elif-else es potente, pero solo es apropiado usarla cuando necesitamos que solo se supere una prueba. En cuanto Python encuentra una prueba que se supera, omite las demás pruebas. En otros casos conviene comprobar todas las condiciones de interés. En esos casos conviene usar una serie de sentencias 'if' simples sin bloques elif ni else.

Ejemplo de lo visto hasta ahora: Comprobar si una lista no está vacía.

```
requested_toppings = []

if requested_toppings: # Devolverá True si la lista no está vacía.
    for requested_topping in requested_toppings:
        print(f"Añadiendo {requested_topping}.")
    print("\nPizza terminada.")
else:
    print("¿Está seguro de que quiere una pizza básica?")
```

Norma de estilo PEP8 en las sentencias 'if': 'if age < 4:' Es mejor que 'if age<4:'aunque ambos funcionan.

DICCIONARIOS.

Un diccionario es similar a una lista, pero nos permite conectar informaciones relacionadas. Por ejemplo, un diccionario que representa a una persona, guarda toda la información sobre la persona: nombre, edad, profesión, etc. También se puede almacenar dos tipos de información que se puedan relacionar, como una lista de palabras y sus significados.

Un diccionario en Python es una colección de pares clave-valor. Cada clave se conecta con un valor y podemos usar una clave para acceder al valor asociada a la misma.

El valor de una clave puede ser un número, un string, una lista, o incluso otro diccionario. De hecho, podemos usar cualquier objeto que creemos en Python como valor en un diccionario.

En Python un diccionario va entre llaves '{}', con una serie de pares clave valor entre ellas. Cuando damos una clave, Python devuelve el valor asociado a ella. Cada clave se conecta con su valor mediante dos puntos y varias claves se separan mediante comas.

Definición de un diccionario.

```
alien_0 = {'color': 'green', 'points': 5}
print(alien_0['color'])
print(alien_0['points'])
```

El diccionario más sencillo: solo un par clave-valor.

```
#alien_0 = {'color': green} # Error: NameError: name 'green' is not defined.
alien_0 = {'color': 'green'}
```

Añadiendo nuevos pares clave-valor.

Los diccionarios son estructuras dinámicas y podemos añadirles nuevos pares de clave-valor en cualquier momento.

```
alien_0 = {'color': 'green', 'points': 5}
print(alien_0)

alien_0['x-position'] = 0
alien_0['y-position'] = 25
print(alien_0)
```

Nota: Los diccionarios mantienen el orden en el que se definen. Cuando se imprima un diccionario o se pasen sus elementos por un bucle, estos se encontraran en el mismo orden en el que fueron añadidos.

Diccionario vacío:

```
alien_0 = {}

alien_0['color'] = 'green'
alien_0['points'] = 5

print(alien_0)
```

Los diccionarios no son inmutables, podemos cambiar el valor de una clave.

```
alien_0['color'] = 'red'
print(alien_0)
```

Ejemplo. Cambiar la posición de un alien en un juego:

```
alien_0 = {'x-position': 0, 'y-position': 25, 'speed': 'medium'}
print(f"La posición x original del alien es: {alien_0['x-position']}")

if alien_0['speed'] == 'slow':
    x_increment = 1
if alien_0['speed'] == 'medium':
    x_increment = 2
else:
    x_increment = 3

alien_0['x-position'] = alien_0['x-position'] + x_increment

print(f"La nueva posición del alien es: {alien_0['x-position']}")
```

Eliminar pares clave-valor.

```
alien_0 = {'color': 'green', 'points': 5}
del alien_0['points']

print(alien_0) # Devuelve: {'color': 'green'}.
```

Representación de diccionarios extensos:

```
favorite_lenguages = {
    'juan': 'c',
    'alberto': 'rust',
    'jaime': 'python',
    'javier': 'c++', # Hemos incluido también una coma.
}
```

Conviene incluir una coma después del último par, así se quedará preparado para añadir otro par en la siguiente línea de código si hace falta.

Usar el método get() para acceder a valores.

Usar la clave para recuperar el valor que nos interesa puede causar un problema en el caso de que la clave que pidamos no exista.

```
alien_0 = {'color': 'red', 'speed': 'medium'}
#print(alien_0['points']) # Error: KeyError: 'points'.
```

Podemos usar el método get() para configurar un valor determinado que se devuelva si la clave no existe. El método get() requiere una clave como argumento, como segundo argumento opcional, podemos pasar el valor que se devolverá si la clave no existe.

```
point_value = alien_0.get('points', 'No existen puntos asignados al alien.')
print(point_value) # Devuelve: "No existen puntos asignados al alien.".
```

Si la clave 'points' existe en el diccionario, obtendremos su valor. Si no existe obtendremos el valor predeterminado. Si dejamos fuera el segundo argumento en la llamada a get() y la clave no existe, Python devolverá el valor 'None', que es un valor especial que indica la ausencia de valor.

Si cabe la posibilidad de que la clave no exista, se ha de considerar usar el método get() en lugar de la anotación por corchetes.

Pasar un bucle por un diccionario.

Podemos pasar por todos los pares clave-valor de un diccionario, solo por las claves o solo por los valores.

MUY IMPORTANTE: Pasar un bucle por todos los pares clave-valor.

```
user_0 = {
    'username': 'pnunez',
    'first': 'pedro',
    'last': 'nuñez',
}

for key, value in user_0.items():
    print(f"\nKey: {key}")
    print(f"Value: {value}")
```

IMPORTANTE: Para escribir un bucle for para un diccionario, creamos nombres para las dos variables que contendrán la clave y el valor de cada par. Podemos elegir los nombres que queramos para estas variables incluso, como es habitual 'k,v':

```
for k, v in user_0.items():
    print(f"{k}, {v}")
```

La segunda mitad de la sentencia for incluye el nombre del diccionario seguido del método items(), que devuelve una lista de pares clave-valor. Después el bucle asigna cada uno de estos pares clave-valor a las dos variables proporcionadas.

Pasar bucle por todas las claves del diccionario. Método keys().

Usamos el método keys() cuando no necesitamos extraer los valores del diccionario si no solamente sus claves.

```
favorite_lenguages = {
    'juan': 'c',
    'alberto': 'rust',
    'jaime': 'python',
    'javier': 'c++',
}
for name in favorite_lenguages.keys():
    print(name.title())
```

Pasar en bucle por las claves es realmente el comportamiento predeterminado al pasar en bucle por un diccionario, así que este código es idéntico en su salida a este otro:

```
for name in favorite_lenguages:
    print(name.title())
```

El método keys(), como cualquier método, no es solo para bucles: en realidad devuelve una lista de todas las claves.

Ejemplo con lo visto hasta ahora:

```
favorite_lenguages = { # Declaramos un diccionario (clave-valor).
    'juan': 'c',
    'alberto': 'rust',
    'jaime': 'python',
    'javier': 'c++',
}
friends = ['alberto', 'javier'] # Declaramos una lista.
for name in favorite_lenguages: # for name in favorite_lenguages.keys():
    print(f"Hola, {name.title()}")

    if name in friends: # Si la clave se encuentra en la lista.
        language = favorite_lenguages[name].title() # [*].
        print(f"{name.title()}, veo que te gusta {language}")
```

En [*] Determinamos el lenguaje favorito de la persona usando el nombre del diccionario y el valor actual de name como clave.

```
print(favorite_lenguages['alberto']) # Devuelve 'rust'.
#print(favorite_lenguages['rust']) # Error: KeyError: 'rust'.
```

Ordenar los resultados devueltos:

```
for name in sorted(favorite_lenguages.keys()): # Uso de la función  sorted().
    print(name.title())
```

Pasar un bucle por todos los valores del diccionario:

Cuando lo que nos interesa son los valores del diccionario y no las claves, usamos el método values() para obtener la lista de valores con claves.

```
for language in favorite_lenguages.values(): # Uso del método values().
    print(language) # Devuelve una lista de los lenguajes.
```

En el enfoque anterior, si hay lenguajes repetidos (valores repetidos en el diccionario) se imprimirían sus repeticiones. Podemos hacer uso de la función set() para sacar los valores únicos, no repetidos.

```
favorite_lenguages = { # Declaramos un diccionario (clave-valor).
    'juan': 'c',
```

```
    'alberto': 'rust',
    'jaime': 'python',
    'javier': 'c++',
    'agustin': 'python'
}

for language in favorite_lenguages.values(): # Lista con valores repetidos.
    print(language)

print("\nLista de lengujes sin repeticiones:\n")
for language in set(favorite_lenguages.values()): # Lista de valores únicos.
    print(language)
```

Definición de conjunto.

Con la función set() hemos usado lo que definimos como conjunto. Un conjunto es una colección en la que cada elemento debe ser único.

```
languages = {'python', 'ruby', 'c', 'rust'} # Declaración de un conjunto.
print(languages)
```

Es fácil confundir un conjunto con un diccionario, ambos usan llaves. Cuando hay llaves, pero no pares clave-valor, se trata de un conjunto.

Repasemos:

Lista: []; Tupla: () e inmutable; Diccionario: {clave-valor}; Conjunto: {}.

Anidación de tipos.

A veces es necesario almacenar múltiples diccionarios en una lista o una lista como valor de un diccionario. La anidación es una característica potente.

Una lista de diccionarios.

```
alien_0 = {'color': 'green', 'points': 50}
alien_1 = {'color': 'red', 'points': 10}
alien_2 = {'color': 'blue'}

aliens = [alien_0, alien_1, alien_2]
print(aliens) # Devuelve: [{'color': 'green', 'points': 50}, {'color': 'red',
              # 'points': 10}, {'color': 'blue'}].
for alien in aliens:
    print(alien)
```

Ejemplo práctico:

```
aliens = [] # Lista vacía para incorporar aliens.

for alien_number in range(30): # Incorpora 30 aliens a la lista.
    new_alien = {'color': 'green', 'points': 5, 'speed': 'slow'}
    aliens.append(new_alien)

for alien in aliens[:5]: # Imprime los 5 primeros aliens.
    print(alien)

print(f"El número total de aliens es: {len(aliens)}") # Imprime total de aliens.
```

También podemos anidar una lista dentro de un diccionario o un diccionario dentro de otro diccionario.

Diccionarios anidados.

```
users = {
    'aeinstein': {
        'first': 'albert',
        'last': 'einstein',
        'location': 'princeton',

    },
    'mcurie': {
        'first': 'marie',
        'last': 'curie',
        'location': 'paris',
    },
}
for username, user_info in users.items():
    print(f"\nUsername: {username}")

    full_name = f"{user_info['first']} {user_info['last']}"
    location = user_info['location']

    print(f"\tFull name: {full_name.title()}")
    print(f"\tLocation: {location.title()}")
```

ENTRADA DEL USUARIO Y BUCLES WHILE

La función input().

La función input() pausa el programa y espera a que el usuario introduzca texto. Una vez que Python recibe la entrada del usuario, la asigna a una variable con la que podemos trabajar.

```
message = input("Dime algo y lo repetiré: ")
print (message) # Imprime la frase y espera la entrada del usuario.
```

La función input(), como puede verse, tiene como argumento el mensaje que se le manda al usuario solicitando la información.

Escribir instrucciones de más de una línea.

```
prompt = "Si me dices quién eres, podemos personalizar el mensaje que ves."
prompt += "\n¿Cuál es tu nombre?"

message = input(prompt)
print(f"\nHola, {message}")
```

La función init().

Usar int() para aceptar entrada numérica.

Si solicitamos un número al usuario, cuando este lo introduzca se almacenará como un string. Debemos hacer el paso de string a representación numérica.

```
age = input("¿Cuál es tu edad?)") # Imprime: ¿Cuál es tu edad? - Respuesta: 21.
age = int(age) # Convertimos el string en número.
print(age >= 18) #Realizamos una comparación con número en lugar de un string.
```

El bucle while.

El bucle while se ejecuta mientras se cumpla una condición. Por ejemplo, podemos usar un bucle while para contar una serie de números.

```
current_number = 1

while current_number <= 5:
    print(current_number)
    current_number += 1 # Abreviación de: current_number = current_number + 1.
```

Ejemplo. Ejecutar un programa hasta que el usuario introduzca 'exit':

```
prompt = "\nDime algo y lo repetiré para ti:"
prompt += "\nEscriba 'exit' para salir.\n"
message = "" # String vacío. De este modo while tiene algo que comprobar en
             # su primera ejecución. IMPORTANTE: en el caso de no declarar esta
             # cadena vacía, while no tendría nada que comparar y el programa
             # terminaría. Error: NameError: name 'message' is not defined. Para
             # evitar esto le damos el valor de cadena vacía.
```

```
while message != 'exit':
    message = input(prompt)
    print(message)
```

Mejorando el programa para que no imprima 'exit'.

```
prompt = "\nDime algo y lo repetiré para ti:"
prompt += "\nEscriba 'exit' para salir.\n"
message = ""

while message != 'exit':
    message = input(prompt)
    if message != 'exit': # Solo se imprime si no coincide con 'exit'.
        print(message)
```

IMPORTANTE: Uso de banderas en while. Flags.

Flag: Si pueden producirse distintos eventos que detengan un bucle while, intentar probar todas esas condiciones en la sentencia while es complicado. Para un bucle while que debería ejecutarse mientras se cumplan varias condiciones, podemos definir una variable que determine si el bucle debe ejecutarse o no. Esta variable, llamada bandera (flag), actúa como una señal para el bucle. Por ejemplo, podemos escribir un programa que se ejecute mientras la bandera sea True y que se detenga cuando cualquier evento cambie la bandera a False. De este modo while necesita comprobar una única condición, la bandera. En otras partes del programa podemos cambiar el estado de la bandera de True a False.

```
prompt = "\nDime algo y lo repetiré para ti:"
prompt += "\nEscriba 'exit' para salir.\n"
active = True # Flag.

while active:
    message = input(prompt)

    if message == 'exit':
        active = False
    else:
        print(message)
```

Usar break para salir del bucle.

```
prompt = "\n Por favor indique el nombre de las ciudades que haya visitado"
prompt += "\n(Introduzca 'exit' cuando haya terminado)\n"

while True: # Ejecución perpetua del bucle hasta llegar a break.
    city = input(prompt)

    if city == 'exit':
        break
    else:
        print(f"Yo también tengo que visitar {city.title()}")
```

Break se puede usar en cualquier bucle de Python. Por ejemplo, puedo usarse en un bucle for que trabaje con una lista o diccionario.

Usar 'continue' en un bucle.

Podemos usar la sentencia 'continue' para volver al principio de un bucle en función del resultado de una prueba condicional.

Ejemplo. Contar del 1 al 10 pero imprimir solo los números impares del rango.

```
number = 0
while number < 10:
    number += 1
    if number % 2 == 0:
        continue
    else:
        print(number)
```

Usar un bucle while con listas y diccionarios.

Un bucle for es efectivo para pasar por una lista, pero no conviene modificar una lista dentro de un bucle for porque Python tendrá problemas para hacer el seguimiento de los elementos de la lista. Si hay que modificar una lista cuando se trabaja con ella, es mejor usar un bucle while.

Emplear bucles while con listas y diccionarios nos permite recoger, almacenar y organizar muchas entradas para examinarlas e informar luego.

Pasar elementos de una lista a otra.

```
unconfirmed_users = ['juan', 'pedro', 'jaime']
confirmed_users = []

while unconfirmed_users:
    current_user = unconfirmed_users.pop()

    print(f"Verificando usuario: {current_user.title()}")
    confirmed_users.append(current_user)

print("\nLos siguientes usuarios han sido confirmados:")
for confirmed_user in confirmed_users:
    print(confirmed_user.title())

# Eliminar todos los casos de valores específicos de una lista.

pets = ['perro', 'gato', 'conejo', 'pájaro', 'gato']
print(pets)

while 'gato' in pets:
    pets.remove('gato')

print(pets) # Imprime: ['perro', 'conejo', 'pájaro'].
```

FUNCIONES

Las funciones nos permiten descomponer un programa en partes pequeñas, cada una con una misión específica. Podemos llamar a una función todas las veces que queramos y guardarla en un archivo aparte. Se puede escribir código que se pueda reutilizar en otros programas.

Las funciones son un bloque de código con un nombre, diseñados para hacer una función específica.

Las funciones se guardan en archivos separados, denominados módulos, para tener organizados los archivos de programa principales.

Definir una función.

```
def greet_user(): # Palabra clave def para definir una función.
    """Muestra un simple saludo."""
    print("¡Hola!")

greet_user() # Llamada a la función greet_user.
```

(Muestra un simple saludo) es un texto conocido como cadena de documentación, que describe lo que hace la función. Las cadenas de documentación van entre triples comillas que es lo que busca Python cuando genera documentación para las funciones de un programa.

Pasar información a una función.

```
def greet_user(user):
    """Muestra un simple saludo."""
    print(f"¡Hola, {user.title()}!")

greet_user('josé')
greet_user('sara')
```

Argumentos y parámetros.

La variable 'user' en la definición de greet_user(user) es un ejemplo de parámetro, una información que necesita la función para hacer su trabajo. Los valores 'josé' y 'sara' son ejemplos de argumentos. Un argumento es una información que se pasa desde una función para llamar a una función, es decir, en este caso asignamos el valor del argumento al parámetro 'user'.

Pasar argumentos.

Igual que la definición de una función puede tener varios parámetros, la llamada a una función puede tener varios argumentos. Existen argumentos posicionales, que tienen que estar en el mismo orden que se escribieron; parámetros de palabra clave, donde cada argumento consta de un nombre de variable y un valor; y listas y diccionarios de valores.

Argumentos posicionales.

Python debe asociar cada argumento de la llamada a la función con un parámetro de la definición. La forma más fácil de hacerlo se basa en el orden de los argumentos proporcionados. Los valores asociados así se denominan "argumentos posicionales".

```
def describe_pet(animal_type, pet_name):
    """Muestra información sobre una mascota."""
    print(f"\nTengo un {animal_type}.")
    print(f"El nombre de mi {animal_type} es {pet_name.title()}.")

describe_pet('gato', 'puka')
```

Argumentos de palabra clave.

Un argumento de palabra clave es un par nombre-valor que pasamos a una función. Los argumentos de palabra clave nos evitan tener que preocuparnos por el orden correcto de los argumentos en la llamada a la función y aclaran el papel de cada valor en la llamada.

Ejemplo:

```
def describe_pet(animal_type, pet_name):
    """Muestra información sobre una mascota."""
    print(f"\nTengo un {animal_type}.")
    print(f"El nombre de mi {animal_type} es {pet_name.title()}.")

describe_pet(animal_type='gato', pet_name='puka')
describe_pet(pet_name='puka', animal_type='gato') # El orden no importa.
```

Valores predeterminados.

```
def describe_pet(pet_name, animal_type='perro'): # Valor predeterminado.
    """Muestra información sobre una mascota."""
    print(f"\nTengo un {animal_type}.")
    print(f"El nombre de mi {animal_type} es {pet_name.title()}.")

describe_pet(pet_name='tor') # Se usa el valor predeterminado.
describe_pet('tor') # También se puede hacer la llamada a la función así.
describe_pet(pet_name='puka', animal_type='gato')
```

Si se proporciona un argumento para un parámetro en la llamada a la función, Python usa el valor del argumento. Si no, utiliza el valor predeterminado del parámetro.

IMPORTANTE: Cuando se usan valores predeterminados, debe colocar cualquier parámetro con un valor predeterminado después de todos los parámetros que no tienen valores predeterminados. Esto permite seguir interpretando argumentos posicionales correctamente.

Valores de retorno.

En ocasiones, las funciones no muestran salida por pantalla, si no que devuelven un valor o un conjunto de valores. Este valor recibe el nombre de valor de retorno. La sentencia return coge un valor dentro de una función y lo devuelve a la línea que ha llamado a la función.

```
def get_formatted_name(first_name, last_name):
    """Devuelve un nombre completo, con formato adecuado."""
    full_name = f"{first_name} {last_name}"
    return full_name.title()
```

```
musician = get_formatted_name('ismael', 'serrano')
print(musician)
```

Hacer un argumento opcional.

Podemos requerir un argumento en opcional en una función. Para ello usamos valores predeterminados, de este modo convertimos un argumento en opcional.

```
def get_formatted_name(first_name, last_name, middle_name=''):
    """Devuelve un nombre completo, con formato adecuado."""
    if middle_name:
        full_name = f"{first_name} {middle_name} {last_name}"
    else:
        full_name = f"{first_name} {last_name}"
    return full_name.title()

musician = get_formatted_name('ismael', 'serrano')
print(musician)

musician = get_formatted_name('alejandro', 'álvarez', 'maría')
print(musician)
```

Tenemos que asegurarnos de que el segundo nombre sea el último argumento pasado para que Python haga coincidir los argumentos posicionales correctamente, como ya habíamos visto.

Devolver un diccionario.

```
def build_person(frist_name, last_name, age=None): # age: argumento opcional.
        """Devuelve un diccionario de información sobre una persona"""
        person = {'frist': frist_name, 'last': last_name}
        if age:
            person['age'] = age
        return person
musician = build_person('ismael', 'serrano', 45)
print(musician) # Imprime: {'frist': 'ismael', 'last': 'serrano', 'age': 45}.
```

En este caso el parámetro opcional tiene el valor por defecto 'None', que se usa cuando una variable no tiene valor específico asignado. En las pruebas condicionales None se evalúa como False.

Usar una función dentro de un bucle while.

Podemos tener un bucle while dentro de una función. También podemos pasar listas y diccionarios en el interior de una función.

```
def get_formatted_name(first_name, last_name):
    """Devuelve un nombre completo, con formato adecuado"""
    full_name = f"{first_name} {last_name}"
    return full_name.title()

while True:
    print("\nDime tu nombre:")
    print("(Introduzca 'q' para salir)")
```

```
    f_name = input("Nombre de pila: ")
    if f_name == 'q':
        break

    l_name = input("Apellido: ")
    if l_name == 'q':
        break

    formatted_name = get_formatted_name(f_name, l_name)
    print(f"\n¡Hola, {formatted_name}!")
```

Pasar una lista a una función.

Cuando pasamos una lista a una función, la función obtiene acceso directo al contenido de la lista. La lista puede estar formada por cualquier estructura de datos, incluso por un diccionario.

```
def greet_users(names):
    """Imprime un saludo sencillo para cada usuario de la lista"""
    for name in names:
        msg = f"Hola, {name.title()}"
        print(msg)
usernames = ['Sara', 'Juan', 'Mónica']
greet_users(usernames)
```

Modificar una lista con una función.

Cualquier cambio que se realice a una función dentro de una lista es permanente. Esto nos permite trabajar con eficiencia cuando tratamos grandes cantidades de datos.

Consideremos el siguiente ejemplo donde no se usan funciones:

```
# Empieza con unos diseños que hay que imprimir.
unprinted_designs = ['carcasa', 'lapicero', 'pelota']
completed_models = []

# Simula la impresión de cada diseño hasta que no queda ninguno.
# Mueve cada diseño a completed_models después de la impresión.
while unprinted_designs:
    current_design = unprinted_designs.pop()
    print(f"Imprimiendo modelo: {current_design}")
    completed_models.append(current_design)

# Muestra todos los modelos completados.
print("\nLos siguientes modelos han sido impresos:")
for completed_model in completed_models:
    print(completed_model)
```

IMPORTANTE: Podemos reorganizar este código escribiendo dos funciones, cada una con una tarea específica. Escribiendo funciones hacemos el código más estructurado. Cada función se ocupa de una parte fundamental del programa.

```
def print_models(unprinted_designs, completed_models):
    """
    Simula imprimir cada diseño, hasta que no queda ninguno.
```

```
    Mueve cada diseño a completed_models después de la impresión.
    """
    while unprinted_designs:
        current_design  = unprinted_designs.pop()
        print(f"Imprimiendo modelo: {current_design}")
        completed_models.append(current_design)

def show_completed_models(completed_models):
    """Muestra todos los modelos que se han imprimido"""
    print("\nLos siguientes modelos han sido impresos:")
    for completed_model in completed_models:
        print(completed_model)
```

Aquí comenzaría la parte principal del programa, mucho más fácil de entender.

```
unprinted_designs = ['carcasa', 'lapicero', 'pelota']
completed_models = []

print_models(unprinted_designs, completed_models)
show_completed_models(completed_models)
```

Esta versión es más fácil de ampliar y mantener que la versión sin funciones. Si necesitamos imprimir más diseño más adelante, podemos simplemente llamar a print_models() otra vez. Si nos damos cuenta de que hay que modificar el código de impresión, podemos cambiarlo una sola vez y los cambios se aplicarán siempre que se llame a la función.

Siempre podemos llamar a una función desde otra función, lo cual puede ser muy útil cuando se descompone una tarea compleja en una serie de pasos.

Evitar que una función modifique una lista.

Para evitar que una función modifique una lista, podemos pasar una copia de la lista en lugar de la original. Supongamos que en el ejemplo anterior queremos conservar la lista unprinted_design sin modificaciones. En ese caso haríamos como sigue: function_name(list_name[:]). Así se envía una copia de la lista a la función. La notación [:] hace una copia de la lista y se la envía a la función. Si no quisiéramos vaciar la lista de diseños sin imprimir, podríamos llamar a print_models() así:

```
print_models(unprinted_designs[:], completed_models) # Llamada a la función.
```

Pasar un número arbitrario de argumentos a una función. (*argumentos).

```
def make_pizza(*toppings):
    """Imprime una lista de los ingredientes seleccionados"""
    print(toppings)

make_pizza('peperoni') # Se pasa un solo argumento.
make_pizza('cebolla', 'carne', 'peperoni') # Se pasan tres argumentos.
```

El parámetro '*toppigns' recoge todos los argumentos que le proporcione la línea de llamada.

El asterisco en el nombre del parámetro *toppigns dice a Python que haga una tupla vacía llamada toppigns y meta ahí cualquier valor que reciba.

Mezclar argumentos posicionales y arbitrarios.

Si quiere que una función acepte distintos tipos de argumentos, debe colocar el parámetro que acepta un número arbitrario de argumentos al final de la definición de la función. Python asocia primero los argumentos posicionales y de palabra clave y luego recoge los restantes en el parámetro final. Por ejemplo:

```
def make_pizza(size, *toppigns):
    """Resume la pizza que estamos a punto de hacer"""
    print(f"\nElaborando una pizza de {size} cmts. con los ingredientes:")
    for toppign in toppigns:
        print(f" - {toppign}")

make_pizza(16, 'peperoni')
make_pizza(12, 'cebolla', 'carne', 'tomate', 'albahaca', 'pimiento')
```

IMPORTANTE: Con frecuencia verá el nombre de parámetro genérico *args, que recoge argumentos posicionales arbitrarios, como éste.

Usar argumentos de palabra clave arbitrarios.

```
def build_profile(first, last, **user_info):
    """Crea un diccionario con todo lo que sabemos sobre un usuario"""
    user_info['first_name'] = first
    user_info['last_name'] = last
    return user_info

user_profile = build_profile('isamel', 'serrano',
                             location = 'argentina',
                             field = 'musica',
                             age = '47',
                             )

print(user_profile)
```

La definición de build_profile() espera un nombre y un apellido y deja que el usuario pase después todos los pares clave-valor que quiera. El doble asterisco ** antes del parámetro **user_info hace que Python cree un diccionario vacío llamado user_info para meter en el cualquier argumento clave-valor que reciba.

IMPORTANTE: Con frecuencia verá el nombre de parámetro genérico **kargs, que recoge argumentos de palabra clave no específicos, como éste.

Guardar las funciones en módulos.

Las funciones se pueden guardar en un archivo llamado módulo y ser importadas mediante al sentencia import al programa principal. import le dice a Python que ponga el código de un módulo a disposición del archivo de programa en ejecución.

Guardar las funciones en un archivo aparte nos permite ocultar los detalles del código del programa para concentrarnos en su lógica de nivel superior.

Importar un módulo completo.

Guardamos las funciones en un archivo llamado modulo.py

Cualquier función disponible en modulo.py estará disponible en el programa en ejecución desde el que hemos declarado 'import modulo'. Esto hace que todas las funciones del módulo estén disponibles en el programa.

Para acceder a las funciones escribiremos: nombre_modulo.nombre_función(). Por ejemplo, si tuviéramos un módulo de funciones llamado pizza.py en el que tenemos la función make_pizza(), accederíamos así a la función:

```
pizza.make_pizza('cebolla', 'carne')
```

Importar funciones específicas.

```
from nombre_módulo import nombre_función.
from pizza import make_pizza
from nombre_módulo import función_0, función_1, función_2, ...
```

Con esta sintaxis no necesitamos la notación del punto al llamar a una función. Como hemos importado explícitamente la función make_pizza en la sentencia import podemos llamarla por su nombre cuando la usemos.

```
make_pizza('cebolla', 'carne')
```

Si el nombre de una función que vamos a importar puede generar un conflicto, podemos usar un alias corto y único haciendo uso de 'as'. Por ejemplo, podemos importar la función make_pizza() como mp() del siguiente modo:

```
from pizza import make_pizza as mp.
```

Usar as para dar un alias a un módulo.

```
"""modules.pizza es una forma de indicar la ruta al módulo. El módulo pizza se encuentra en el directorio modules. alias pizza para modules.pizza."""
import modules.pizza as pizza

pizza.make_pizza('15', 'peperoni')

from modules.pizza import make_pizza as mp # alias mp para make_pizza

make_pizza('15', 'pollo') # Ahora no es necesaria la notación del punto.
mp('12', 'carne') # Al tener un alias podemos llamar a la función de este modo.
```

Podemos importar todas las funciones de un módulo con el operador (*). De este modo no será necesario utilizar la notación del punto para llamar a las funciones:

```
from modules.pizza import *

make_pizza('20', 'carne picada') # No es necesaria la notación del punto.
```

Es mejor no usar esta técnica al trabajar con módulos grandes que no haya escrito usted. Si el módulo tiene un nombre de función que coincida con otro de su proyecto, puede obtener resultados inesperados.

IMPORTANTE: Lo mejor es importar la función o funciones que le interesen, o el módulo completo, y usar la notación de punto. Así el código queda más claro y es más fácil de leer.

Dar estilo a las funciones.

Si especifica un valor para un parámetro, no debería usar espacios a los lados del signo de igual.

```
def nombre_función(parámetro_0, parámetro_1='valor predeterminado')
```

La misma convección se aplica a los argumentos de la palabra clave en las llamadas a una función:

```
nombre_función(valor_0, parámetro_1='valor')
```

PEP8 para definición de funciones.

Si la declaración de parámetros supera los 79 caracteres, se escribirán de la siguiente forma:

```
def function(
    parameter_0, parameter_1, parameter_2,
    parameter_3, parameter_4, parameter_5):

    print("")
```

Si su módulo o programa incluye más de una función, puede separarlas con dos líneas en blanco para dejar más claro dónde acaba una y empieza la otra.

Todas las sentencias import deben escribir al principio del archivo, con la única excepción de si hay comentarios iniciales para describir el programa completo.

Alcance (scope) de los nombres.

En todos los casos, el alcance de una variable (donde se puede usar) siempre está determinado por dónde se asigna en su código fuente y no tiene nada que ver con qué funciones llaman a qué. De hecho, las variables se pueden asignar en tres lugares diferentes, correspondientes a tres ámbitos diferentes:

1. Si se asigna una variable dentro de una definición def, es local para esa función.

2. Si se asigna una variable en una definición def que anida a otras, no es local para las funciones anidadas.

3. Si se asigna una variable fuera de todas las definiciones def, es global para todo el archivo.

Por ejemplo, en el siguiente código, la asignación X = 99 crea una variable global llamada X (visible en todas parte del archivo al que pertenece), pero la asignación X = 88 dentro de def crea un variable local X (visible solo dentro de la instrucción def). Aunque ambas variables se denominan X , sus alcances (espacio de nombres) las hacen diferentes. El efecto es que los ámbitos de función ayudan a evitar conflictos de nombres en sus programas y ayudan a hacer que las funciones sean unidades de programa más independientes.

```
X = 99
def func():
    X = 88
```

Las funciones tienen un ámbito local y los módulos definen un ámbito global. Las variables del módulo se convierten en atributos de un objeto del módulo para con el exterior al módulo, pero se pueden usar como variables simples dentro del módulo.

El alcance global abarca un solo archivo, no hay que dejarse engañar por la palabra "global"; los nombres en el nivel superior de un archivo solo son globales para ese archivo. Cuando escuchemos global en Python, debemos pensar en un módulo.

Los nombres asignados son locales a menos que se declaren globales o no locales. Por defecto, todos los nombres asignados dentro de una definición de función se colocan en el ámbito local (el espacio de nombres asociado con la llamada de la función). Si necesita asignar un nombre que vive en el nivel superior del módulo que encierra la función, puede hacerlo declarándolo como global dentro de la función. Si necesita asignar un nombre que vive en una definición adjunta, puede hacerlo declarándolo como 'nonlocal'.

```
# Alcace 'global'.
X = 99                          # X y func tienen alcance global.

def func(Y):                    # Y y Z tienen alcance local.
    # Alcance local.
    Z= X + Y                    # X es global.
    return Z

print(func(1))                  # Imprime: 100.
```

La declaración de un nombre como global nos permite cambiar los nombres que viven fuera de una definición de función, en el nivel superior del módulo. Veamos un ejemplo:

```
X = 88                          # X es global.

def func():
    global X                    # X es global, fuera de func()
    X = 99

func()
print(X)                        # Imprime: 99.
```

La declaración 'nonlocal' es prima cercana de global; a diferencia de 'global', 'nonlocal' se aplica a un nombre en el ámbito de una función envolvente, no al alcance global del módulo. Si se usa 'nonlocal', significa que Python, buscará una variable con el mismo nombre desde un ámbito superior.

Alcance 'nonlocal'.

A continuación, se construye y devuelve la función anidada, que es llamada más tarde; en la referencia de estado en los mapas anidados, el alcance local, está usando las reglas normales de búsqueda de alcance:

```
def tester(start):
    state = start
    def nested(label):
        print(label, state)
    return nested

F = tester(0)
print(F('spam'))
print(F('ham'))
```

Sin embargo, cambiar un nombre en el ámbito de una definición adjunta no está permitido de forma predeterminada, como podemos ver en el siguiente ejemplo:

```
def tester(start):
    state = start
    def nested(label):
        print(label, state)
        #state += 1#Error (No está permitido): UnboundLocalError: local variable
                   #'state' referenced before assignment.
    return nested

F = tester(0)
print(F('spam'))
print(F('ham'))
```

Podemos hacer uso del alcance 'nonlocal' para los cambios. Ahora, si declaramos el estado en el alcance como no local dentro de la función anidada, también podemos cambiar el nombre dentro de dicha función; de este modo:

```
def tester(start):
    state = start
    def nested(label):
        nonlocal state
        print(label, state)
        state += 1
    return nested

F = tester(0)
print(F('spam'))
print(F('ham'))
```

CLASES

Las clases combinan funciones y datos en un paquete limpio que se puede usar de forma flexible y eficiente.

Una clase representa un objeto y situación del mundo real. Cuando escribimos una clase, definimos el comportamiento general que puede tener una categoría completa de objetos. Ahora, al crear un objeto particular de la clase, este se dota automáticamente del comportamiento general; después podemos dar a cada objeto los rasgos únicos que queramos.

La creación de un objeto a partir de una clase recibe el nombre de instanciación, de este modo, trabajamos con instancias de una clase.

Podemos modelar prácticamente cualquier cosa usando clases. Por ejemplo, la clase Dog puede representar a un perro, pero no a uno en particular, si no a todos los perros. Los perros tienen un nombre y una edad, esos son sus datos, atributos. Muchos perros saben sentarse y hacer la croqueta, esos son sus comportamientos. Los datos nombre y edad y los comportamientos sentarse y hacer la croqueta se usan para definir la clase. Así haremos un objeto que representa a un perro, más tarde se pueden hacer instancias específicas que representen a perros individuales.

Creación de una clase.

Crearemos la clase Dog. Cada instancia creada a partir de la clase Dog contendrá un nombre y una edad y daremos a cada perro la capacidad de sentarse 'sit()' y hacer la croqueta 'roll_over()', estos serán los que reciben el nombre de métodos de la clase.

```
class Dog: # Definimos la clase Dog, inicial mayúscula por convención.
    """Un intento de modelar a un perro"""

    def __init__(self, name, age):
        """Inicializa los atributos de nombre y edad."""
        self.name = name.title() # Capitalización de name. Ver debajo [***]
        self.age = age

    def sit(self):
        """Simula un perro sentándose en respuesta a una orden."""
        print(f"{self.name} está ahora sentándose")

    def roll_over(self):
        """Simula un perro haciendo la croqueta en respuesta a una orden."""
        print(f"{self.name} ¡Hace la croqueta!")
```

Vayamos paso a paso para explicar la creación de la clase.

En la definición de esta clase, class Dog, no hay paréntesis porque se está creando desde cero. Se escribe una cadena de documentación que describe lo que hace la clase.

El método __init__().

El método __init__(). Una función que forma parte de una clase es un método. Todo lo visto sobre funciones se aplica también a los métodos. Este método es un método especial que ejecutará Python automáticamente siempre que creemos una nueva instancia de la clase Dog. Tiene dos guiones bajos a cada lado, es una convención que permite evitar que los nombres de métodos predeterminados de Python entren en conflicto con los nuestros.

El parámetro self.

El parámetro self es necesario en la definición del método y debe ir antes que los otros. self es una referencia a la propia instancia creada a partir de la clase, dando a la instancia individual acceso a los atributos y los métodos de la clase.

Cuando creemos una instancia de Dog, Python llamará al método __init__() desde la clase Dog, pasaremos a Dog() un nombre y una edad como argumentos; self pasa automáticamente, aquí no tenemos que hacer nada. Es decir, cuando queramos hacer una instancia para la clase Dog, solo deberemos pasar los dos últimos parámetros, name y age.

El prefijo self.

Las dos variables definidas tienen el prefijo self: self.name y self.age. Cualquier variable prefijada con self estará disponible para los métodos de la clase y podremos acceder a estas variables a través de cualquier instancia creada desde la clase. Estos se denominan atributos.

Métodos definidos.

La clase Dog tiene otros dos métodos definidos: sit() y roll_over(). Estos métodos no necesitan información adicional para ejecutarse, simplemente los definimos para que tengan un parámetro, self. Las instancias que creemos después tendrán acceso a estos métodos; en otras palabras, serán capaces de sentarse y de hacer la croqueta.

Instanciar una clase.

```
my_dog = Dog('tor', 11)

print(f"El nombre de mi perro es: {my_dog.name}") # Accediendo al atributo name.
print(f"La edad de mi perro es de {my_dog.age} años.")
```

Cuando Python lee la primera línea, llama al método __init__() de Dog con los argumentos pasados. El método __init__(), crea una instancia que representa a este perro en particular y configura los atributos name y age con los valores proporcionados como argumentos. Observar: un nombre con la inicial en mayúscula, Dog, se refiera a una clase y un nombre en minúsculas como my_dog se refiere a una sola instancia creada de una clase.

Para acceder a los atributos de una instancia, usamos la notación de punto. 'my_dog.name'.

Llamadas a métodos.

Una vez creada una instancia, usamos la notación de punto para llamar a cualquier método definido en la clase.

```
my_dog.sit()
my_dog.roll_over()

#[***]
#{my_dog.title()}.sit() # Error:AttributeError:
                        #'Dog'object has no attribute 'title'.
```

Trabajar con instancias y clases.

Conviene aprender cómo modificar los atributos asociados con una instancia en particular. Se pueden modificar directamente o escribiendo métodos que actualicen los atributos de forma específica.

```
class Car:
    """Un simple intento de representar a un coche"""

    def __init__(self, make, model, year):
        """Inicializa atributos para describir el coche"""
        self.make = make.title()
        self.model = model.title()
        self.year = int(year)

    def get_descriptive_name(self):
        """Devuelve un nombre descriptivo con el formato adecuado"""
        long_name = f"{self.year} {self.make} {self.model}"
        return long_name

my_car = Car('mazda', '3', '2006')
print(my_car.get_descriptive_name())
```

Establecer un valor predeterminado para un atributo.

```
class Car:
    """Un simple intento de representar a un coche"""

    def __init__(self, make, model, year):
        """Inicializa atributos para describir el coche"""
        self.make = make.title()
        self.model = model.title()
        self.year = int(year)
        self.odometer_reading = 10_000 # Atributo predeterminado.

    def get_descriptive_name(self):
        """Devuelve un nombre descriptivo con el formato adecuado"""
        long_name = f"{self.year} {self.make} {self.model}"
        return long_name

    def read_odometer(self):
        """Imprime una oración que indica el kilometraje del coche"""
        print(f"Este coche tiene {self.odometer_reading} kilómetros")

my_car = Car('mazda', '3', '2006')
print(my_car.get_descriptive_name())
my_car.read_odometer()
```

Modificar el valor de un atributo.

Modificar el valor de un atributo directamente.

Accedemos directamente al atributo a través de una instancia.

```
my_car.odometer_reading = 15_000
my_car.read_odometer() # Imprime: "Este coche tiene 15000 kilómetros".
```

Modificar el valor de un atributo a través de un método.

En ocasiones conviene escribir un método que actualice el valor del atributo. En lugar de acceder directamente al atributo, pasamos el valor a un método que gestiona la actualización internamente.

```
class Car:
    """Un simple intento de representar a un coche"""

    def __init__(self, make, model, year):
        """Inicializa atributos para describir el coche"""
        self.make = make.title()
        self.model = model.title()
        self.year = int(year)
        self.odometer_reading = 10_000 # Atributo predeterminado.

    def get_descriptive_name(self):
        """Devuelve un nombre descriptivo con el formato adecuado"""
        long_name = f"{self.year} {self.make} {self.model}"
        return long_name

    def read_odometer(self):
        """Imprime una oración que indica el kilometraje del coche"""
        print(f"Este coche tiene {self.odometer_reading} kilómetros")

    def update_odometer(self, mileage):
        """Configura el kilometraje con le valor dado"""
        self.odometer_reading = mileage

my_car = Car('mazda', '3', '2006')
print(my_car.get_descriptive_name())

my_car.update_odometer(20_000)
my_car.read_odometer() # Imprime: "Este coche tiene 20000 kilómetros".
```

Herencia.

No siempre hace falta empezar desde cero para crear una clase. Si la clase que queremos hacer es una versión especializada de otra que hemos escrito, podemos usar la herencia. Cuando una clase hereda de otra, coge los atributos y métodos de la primera. La clase original se llama 'clase base' y la nueva es la 'clase derivada'. La clase derivada puede heredar algunos o todos los atributos y métodos de su clase base, pero también tiene libertad para definir nuevos atributos y métodos propios.

El método __init__() para una clase derivada.

Cuando escribimos una clase basada en otra, con frecuencia, necesitaremos llamar al método __init__() de la clase base. Esto inicializará todos los atributos definidos en el método __init__() de la base y hará que estén disponibles para la derivada.

Crearemos una clase de coche eléctrico derivada de la clase Car. Así solo tendremos que escribir código para los atributos y métodos específicos.

```
class ElectricCar(Car): # IMPORTANTE: (Car)
    """Representa aspectos de un coche propios de los vehículos eléctricos"""
```

```
    def __init__(self, make, model, year):
        """Inicializa los atributos de la clase base"""
        super().__init__(make, model, year)

my_tesla = ElectricCar('tesla', 'model s', '2019')
print(my_tesla.get_descriptive_name())
```

IMPORTANTE: Cuando creamos una clase derivada, la clase base debe formar parte del archivo actual y aparecer antes que la derivada.

Al definir la clase derivada, el nombre de la clase base debe aparecer entre paréntesis en la definición de la clase derivada.

El método __init__() coge la información necesaria para crear una instancia de Car.

La función super() es una función especial que nos permite llamar a un método de la clase base. El nombre super viene de una convención de llamar a la clase base 'superclase' y a la derivada 'subclase'. Esa línea dice a Python que llame al método __init__() de Car, que da a una instancia ElectricCar todos los atributos definidos en ese método.

Definir atributos y métodos para una clase derivada.

```
class ElectricCar(Car): # IMPORTANTE: (Car)
    """Representa aspectos de un coche propios de los vehículos eléctricos"""

    def __init__(self, make, model, year):
        """Inicializa los atributos de la clase base"""
        super().__init__(make, model, year)
        self.battery_size = 75

    def describe_battery(self):
        """Imprime una frase que describe el tamaño de la batería"""
        print(f"Este coche tiene una batería de {self.battery_size} KWh.")

my_tesla = ElectricCar('tesla', 'model s', 2019)
print(my_tesla.get_descriptive_name())
my_tesla.describe_battery() # Imprime: Este coche tiene una batería de 75 KWh.

#my_tesla = ElectricCar('tesla', 'model s', 2019, 90) # No funciona. [*]
```

[*] Error: TypeError: ElectricCar.__init__() takes 4 positional arguments but 5 were given.

Al crear la instancia my_tesla no podemos pasar el argumento de battery_size, en su creación, para cambiar el valor por defecto. Se nos devolverá el error indicado. En su lugar, deberíamos hacerlo del siguiente modo:

```
class ElectricCar(Car):
    """Representa aspectos de un coche propios de los vehículos eléctricos"""

    def __init__(self, make, model, year, batery_size):
        """Inicializa los atributos de la clase base"""
        super().__init__(make, model, year)
        self.battery_size = batery_size

    def describe_battery(self):
```

```
        """Imprime una frase que describe el tamaño de la batería"""
        print(f"Este coche tiene una batería de {self.battery_size} KWh.")

my_tesla = ElectricCar('tesla', 'model s', 2019, 90)
print(my_tesla.get_descriptive_name())
my_tesla.describe_battery() # Imprime: Este coche tiene una batería de 90 KWh.
```

Anular/Sustituir métodos de la clase base.

Podemos anular cualquier método de la clase base que no se ajuste a lo que intentamos modelar con la clase derivada. Para ello, definimos un método en la derivada con le mismo nombre que el método de la base que queremos anular. Python ignorará el método de la clase base y solo prestará atención al método que hemos definido en la clase derivada. Veamos un ejemplo.

```
class ElectricCar(Car):
    """Representa aspectos de un coche propios de los vehículos eléctricos"""

    def __init__(self, make, model, year, batery_size):
        """Inicializa los atributos de la clase base"""
        super().__init__(make, model, year)
        self.battery_size = batery_size

    def describe_battery(self):
        """Imprime una frase que describe el tamaño de la batería"""
        print(f"Este coche tiene una batería de {self.battery_size} KWh.")

    def update_odometer(self): # Sustituimos el método de la clase base 'Car'.
        """Configura el kilometraje con le valor dado"""
        return None # Una función debe devolver el valor esperado o 'None'.

my_tesla = ElectricCar('tesla', 'model s', 2019, 90)
my_tesla.update_odometer() # Esta llamada no hace nada, solo un return.
```

Instancias como atributos (Descomponer una clase en otras).

Podríamos terminar una clase con una lista de atributos y métodos interminable que hiciera que el código se alargara demasiado. Conviene saber que se puede escribir parte de una clase como una clase aparte, es decir, podemos descomponer una clase grande en clases más pequeñas que funcionen juntas.

Supongamos que hemos añadido muchos atributos y métodos en 'ElectriCar' para la batería del coche, nos convendría mandar esos atributos y métodos a una clase nueva llamada 'Battery'. Luego podemos usar una instancia de Battery como atributo de la clase 'ElectricCar'.

```
class Battery:
    """Un simple intento de modelar una batería para un coche eléctrico"""

    def __init__(self, battery_size=75):
        """Inicializa los atributos de la batería"""
        self.battery_size  = battery_size

    def describe_battery(self):
        """Imprime una frase que describe el tamaño de la batería"""
        print(f"Este coche tiene una batería de {self.battery_size} KWh.")
```

```
class ElectricCar(Car):
    """Representa aspectos de un coche propios de los vehículos eléctricos"""

    def __init__(self, make, model, year):
        """
        Inicializa los atributos de la clase base.
        Luego inicializa los atributos específicos de un coche eléctrico
        """
        super().__init__(make, model, year)
        self.battery = Battery() # IMPORTANTE: Instancia como atributo.

my_tesla = ElectricCar('tesla', 'model s', 2019)

print(my_tesla.get_descriptive_name())
```

Cuando usamos el método describe_battery, usamos el atributo battery de la instancia.

```
my_tesla.battery.describe_battery() # IMPORTANTE: atributo battery de my_tesla.
```

Esta última línea le dice a Python que mire la instancia my_tesla, busque su atributo battery y llame al método describe_battery() asociado con la instancia de Battery almacenada en el atributo.

Importar clases.

Python nos permite almacenar las clases en módulos para luego importar las que necesitemos a nuestro programa principal.

Importar una sola clase.

```
from classes.car import Car
```

Esta sentencia le dice a Python que abra el módulo car que se encuentra en la carpeta classes e importa la clase Car.

Importar varias clases desde un módulo.

```
from classes.car import Car, ElectricCar # Las separamos con comas.
```

Importar un módulo entero.

También podemos importar un módulo entero y acceder a las clases necesarias con la notación de punto. Es una aproximación sencilla que tiene como resultado código fácil de leer. Como cada llamada que crea una instancia de una clase incluye el nombre del módulo, no habrá conflictos con los nombres usados en el archivo actual.

```
import classes.car # Importamos el módulo entero (ruta a la carpeta).

my_car = classes.car.Car('mazda', '3', 2006)
my_tesla = classes.car.ElectricCar('tesla', 'model s', 2019, 150)

print(my_car.get_descriptive_name())
print(my_tesla.get_descriptive_name())
```

Usar alias.

Como ya hemos visto los alias son muy útiles cuando usamos módulos para organizar el código de nuestros proyectos. Podemos usarlos también para importar clases.

```
from classes.car import ElectricCar as EC

my_tesla = EC('tesla', 'model s', 2023, 75)
print(my_tesla.get_descriptive_name())
```

La biblioteca estándar de Python.

La 'biblioteca estándar de Python' es un conjunto de módulos incluidos con la instalación de Python. Puede usar cualquier función o clase de la biblioteca estándar simplemente incluyendo una sentencia import al principio de su archivo.

Importando un módulo de la biblioteca estándar.

```
from random import randint
print(randint(1, 6))
```

Dar estilo a las clases.

Los nombres de las clases deberían escribirse en CamelCase, es decir, mayúsculas en la inicial de cada palabra y sin guiones. Los nombres de instancias y módulos deberían escribirse en minúsculas y con guiones bajos entre palabras.

Cada clase y cada módulo deberían tener una cadena de documentación. Este comentario debería seguir las misma convenciones de formato usadas para las cadenas de documentación de las funciones.

Si necesita importar un módulo de la biblioteca estándar y otro suyo, coloque primero la sentencia import para la biblioteca estándar. Después añada una línea en blanco y la sentencia import para su modulo.

ARCHIVOS

Leer un archivo.

Se puede leer todo el contenido de un archivo o pasar por él línea por línea.

```
#Lee el arrchivo misc/pi_digits.txt y lo imprime en pantalla.
with open ('misc/pi_digits.txt') as file_object: # Ruta al archivo.
    contents = file_object.read() # Método red().
print(contents)
```

Para trabajar con un archivo, aunque solo sea para imprimirlo, primero hay que abrirlo para acceder a él, función 'open()'. La función open() devuelve un objeto que representa a un archivo.

La palabra clave with cierra el programa cuando ya no hace falta acceder a él. Fíjese que llamamos a open() en este programa, pero no a close(). Podríamos llamar directamente a close(), pero no es práctico por motivos de posibles pérdidas de datos. Con esta estructura Python cerrará el archivo por nosotros llegado el momento. Python cerrará el archivo cuando el bloque with termine de ejecutarse.

La única diferencia entre la salida y el archivo original es una línea en blanco extra al final de la salida. Esta línea aparece porque read() devuelve una cadena vacía cuando llega al final del archivo; esta cadena se muestra como línea en blanco. Si quiere eliminar esa línea puede usar rstrip() en la llamada a print.

```
print(contents.rstrip())

# Leer línea por línea.

filename = 'misc/pi_digits.txt'

with open (filename) as file_object:
    for line in file_object:
        print(line.rstrip())
```

Hacer una lista de línea de un archivo.

Cuando usamos with, el objeto que representa al archivo que devuelve open() solo está disponible dentro del bloque with que lo contiene. Si queremos mantener el acceso al contenido de un archivo fuera del bloque with, podemos guardar esas líneas del archivo en una lista dentro del bloque y más tarde trabajar con ellas.

```
filename = 'misc/pi_digits.txt'

with open (filename) as file_object:
    lines = file_object.readlines() # Coge cada línea y la guarda en una lista.

print(lines)

for line in lines:
    print(line.strip())
```

Trabajar con el contenido de un archivo.

Ejemplo ilustrativo:

```
filename = ('misc/pi_digits.txt')

with open(filename) as file_object:
    lines = file_object.readlines()

pi_string = ''
for line in lines:
    pi_string += line.strip()

print(pi_string)
print(len(pi_string))
```

IMPORTANTE: pi_string es una cadena no un número. Deberíamos usar el método float() para convertirlo en número.

Escribir un archivo.

```
filename = 'misc/programming.txt'

with open(filename, 'w') as file_object: # Para escribir en un archivo ('w').
    file_object.write("Hola, amo la programación") # Método write.
```

La llamada a open() contiene ahora dos argumentos: el archivo y 'w'. Si omitimos este segundo argumento, Python abre el archivo en modo solo lectura. Las opciones para este segundo argumento son: modo lectura 'r', modo escritura 'w', modo anexo 'a' o un modo que permite leer y escribir 'r+'.

La función open() crea automáticamente, si no existe ya, el archivo en el que vamos a escribir. Hay que tener cuidado cuando se abre un archivo en modo escritura 'w' porque, si existe, Python borrará su contenido antes de devolver el objeto del archivo, es decir, lo sobrescribirá.

IMPORTANTE: Python solo puede escribir cadenas en un archivo de texto. Si quiere almacenar números tendrá que convertir antes los datos a formato de cadena con la función str().

Anexar a un archivo.

Añadir contenido a un archivo en lugar de escribir sobre el contenido existente.

```
filename = 'misc/programming.txt'

with open(filename, 'a') as file_object: # Usamos el argumento 'a': anexar.
    file_object.write("\nEsto es una nueva línea")
```

Almacenar datos.

El módulo json permite volcar estructuras de datos de Python simples en un archivo y cargar los datos desde ahí la próxima vez que se ejecute el programa.

```
print(dir(filename))

# json.dump() y json.load().

import json

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]

filename = 'misc/numbers.json'
with open(filename, 'w') as f:
    json.dump(numbers, f)
```

La función json.dump() toma dos argumentos: un dato que almacenar y un objeto de archivo que pueda usar para guardar el dato.

```
with open(filename) as f:
    numbers = json.load(f)

print(numbers)
```

Guardar y leer datos generados por usuarios.

Ejemplo: el siguiente programa pide al usuario su nombre y apellido cuando se ejecuta por primera vez y luego se recuerdan esos datos en la siguiente ejecución mediante su almacenamiento en un archivo.

```
import json
"""
Guardamos la información del usuario.
"""
username = input("Introduzca su nombre:")

filename = 'misc/username.json'
with open(filename, 'w') as f:
    json.dump(username, f)
    print(f"Recordaremos su nombre cuando regrese, {username}.")

"""
Recuperamos la información del usuario.
"""
filename = 'misc/username.json'

with open(filename) as f:
    username = json.load(f)
    print(f"¡Bienvenido, {username}!")
```

Excepciones.

Python utiliza objetos especiales llamados 'excepciones' para administrar los errores que surjan durante la ejecución de un programa. Siempre que se produzca un error que haga que Python no sepa qué hacer a continuación, crea un objeto de excepción. Si escribimos código que maneje la excepción, el programa seguirá ejecutándose. Si no manejamos la excepción, el programa se detendrá y mostrará un rastreo que incluye un informe de la excepción en cuestión.

Las excepciones se manejan con bloques try-except. Un bloque try-except pide a Python que haga algo, pero también le dice qué hacer si surge una excepción. Con

su uso los programas seguirán ejecutándose incluso si algo se tuerce y, en lugar de rastreos, los usuarios verán los mensajes de error que escribamos para ellos.

Manejar la excepción de división por cero.

```
print (5/0) # Error (excepción: ZeroDivisionError: division by zero).
```

Python ha creado el objeto de excepción: ZeroDivisionError. Podemos usar esta información para modificar nuestro programa. Diremos a Python qué ha de hacer cuando se produzca este tipo de error.

Usar bloques try-except.

```
try:
    print(5/0)
except ZeroDivisionError:
    print("Está intentando dividir por 0")
```

Si el código del bloque try funciona, Python omite el bloque except. Si el código del bloque try provoca un error, Python busca un bloque except cuyo error coincida con el que se ha producido y ejecuta el código de ese bloque.

Un código bien escrito y bien probado no es muy propenso a errores internos. Pero cuando el programa depende de algo externo (usuario, archivos inexistentes, red, ...) cabe la posibilidad de que se produzca una excepción. Con un poco de experiencia sabrá dónde incluir bloques de manejo de excepciones.

El bloque 'else' de try-except.

```
print("Indicar dos números que quiera dividir:")
print("Escriba 'q' para salir")

while True:
    first_number = input("\nPrimer número:")
    if first_number == 'q':
        break
    second_number = input("Segundo número:")
    if second_number == 'q':
        break
    try:
        answer = int(first_number) / int(second_number)
    except ZeroDivisionError:
        print("Está intentando dividir por 0.")
    else:
        print(answer)
```

Manejar la excepción FileNotFoundError.

Un problema habitual cuando trabajamos con archivos es el manejo de archivos que faltan.

```
filename = 'misc/alice.txt'

try:
    with open(filename, encoding='utf-8') as f:
        contents = f.read()
except FileNotFoundError:
    print(f"El archivo {filename} no existe.")
else:
    words = contents.split()
    num_words = len(words)
    print(f"El archivo {filename} tiene {num_words} palabras.")
```

Fallos silenciosos.

No es necesario informar de todas las excepciones que capturemos. A veces nos interesa que el programa falle en silencio cuando se produzca una excepción y siga después como si no hubiese pasado nada. Para escribir esto diremos a Python explícitamente que no haga nada en el bloque except. Python cuenta con la sentencia pass que indicar 'no hacer nada'.

```
filename = 'misc/alice.txt'

try:
    with open(filename, encoding='utf-8') as f:
        contents = f.read()
except FileNotFoundError:
    pass # Aquí no hacer nada: Fallo silencioso.
else:
    words = contents.split()
    num_words = len(words)
    print(f"El archivo {filename} tine {num_words} palabras.")
```

Refactorización.

A menudo llegará a un punto en el que su código funcione, pero se dé cuenta de que podría mejorarlo dividiéndolo en una serie de funciones con tareas específicas. Este proceso recibe el nombre de 'refactorización' y hace que nuestro código sea más limpio, fácil de entender y fácil de ampliar.

Ejemplo: ejemplo de lo visto hasta ahora con un código ya refactorizado.

```
import json

def get_stored_username():
    """Obtiene un nombre de usuario almacenado si está disponible."""
    filename = 'misc/username.json'
    try:
        with open(filename) as f:
            username = json.load(f)
    except FileNotFoundError:
        return None # Una función debe devolver el valor esperado o 'None'.
    else:
        return username

def get_new_username():
    """Solicita un nuevo nombre de usuario."""
    username = input("Introduzca su nombre")
```

```
    filename = 'misc/username.json'
    with open(filename, 'w') as f:
        json.dump(username, f)
        return username

def greet_user():
    """Saluda al usuario por su nombre."""
    username = get_stored_username()
    if username:
        print(f"Hola de nuevo, {username}")
    else:
        username = get_new_username()
        print(f"Hola, te recordaré cuando regreses, {username}")

greet_user()
```

Cada función, en el ejemplo anterior, tiene un único propósito claro. Llamamos a greet_user() y esa función imprime un mensaje apropiado: o da la bienvenida de vuelta a un usuario existente o saluda a uno nuevo. Lo hace llamando a get_stored_username(), que es responsable solo de recuperar un nombre de usuario almacenado si lo hay, Por último, greet_user() llama a get_new_username() si hace falta, que se ocupa de obtener un nuevo nombre de usuario y guardarlo.

Esta división del programa es una parte esencial de escribir código claro que sea fácil de mantener y de ampliar.

PROBAR EL CÓDIGO.

Cuando escribimos una función o una clase, también podemos escribir pruebas para ese código. Se trata de comprobar si el código funciona como debería en respuesta a todos los tipos de entrada que esté diseñado para recibir.

Probar una función.

```
from modules.name_function import get_formatted_name # Importamos una función

print("Introduzca 'q' para salir")
while True:
    first = input("Introduzca su nombre:")
    if first == 'q':
        break
    last = input("Introduzca su apellido:")
    if last == 'q':
        break

    formatted_name = get_formatted_name(first, last)
    print(f"\tSu nombre completo es: {formatted_name}")
```

Supongamos que queremos modificar get_formatted_name para que gestione también nombres compuestos. Al hacerlo, tenemos que asegurarnos de no estropear la forma en la que la función gestiona los nombres que solo constan de un nombre simple y apellido. Podríamos hacerlo de forma repetitiva manualmente, pero sería bastante tedioso.

Python ofrece una manera de automatizar las pruebas de la salida de una función. Si automatizamos las pruebas de get_formatted_name(), siempre podremos confiar en que la función funcionará cuando reciba los tipos de datos para los que hemos escrito pruebas.

Pruebas unitarias y casos de prueba.

El módulo 'unittest' de la biblioteca estándar ofrece herramientas para probar el código. Una 'prueba unitaria' comprueba que un aspecto específico del comportamiento de una función es correcto. Un 'caso de prueba' es un conjunto de pruebas unitarias que juntas comprueban si la función se comporta como debería.

Una prueba que pasa.

Para escribir un caso de prueba para una función, importe el módulo unittest y la función que desee probar. Luego cree una clase que herede de unittest.TestCase y escriba una serie de métodos para comprobar los distintos aspectos del comportamiento de la función.

```
import unittest

from modules.name_function import get_formatted_name

class NamesTestCase(unittest.TestCase):
    """Pruebas para name_function.py."""

    def test_first_last_name(self):
        """¿Funcionan nombres como 'Ismael Serrano'?"""
        formatted_name = get_formatted_name('ismael', 'serrano')
```

```
        self.assertEqual(formatted_name, 'Ismael Serrano')

if __name__ == '__main__':
    #unittest.main()  # Descomentar.
    None              # Comentar
```

Expliquemos paso a paso el código escrito:

Primero, importamos unittest y la función que queremos probar, get_formatted_name().

Creamos una clase 'NamesTestCase' que hereda de 'unittest.TestCase', que contendrá una serie de pruebas unitarias para get_formatted_name().

NameTestCase contiene un único método que prueba un aspecto de get_formatted_name(), con él queremos comprobar que los nombres con solo un nombre y apellido reciben el formato correcto. Observar que hemos llamado al método 'test_first_last_name', cualquier método que empiece por test_ se ejecutará automáticamente. Dentro de este método llamamos a la función que queremos probar.

Usamos una de las características más útiles de unittest: un método assert. Estos métodos comprueban que el resultado recibido coincide con el resultado esperado.

IMPORTANTE: El bloque if busca una variable especial, __name__, que se establece cuando se ejecuta el programa. Si esta archivo se ejecuta como programa principal, el valor de __name__ es '__main__'. En este caso, llamamos a unittest.main(), que ejecuta el caso de prueba. Cuando un marco de pruebas importe este archivo, el valor de __name__ no será '__main__' y este bloque no se ejecutará.

Esta es la salida que obtenemos al ejecutar el código:

```
.
----------------------------------------------------------------------
Ran 1 test in 0.000s
OK
```

Una prueba que falla.

```
import unittest

from modules.name_function import get_formatted_middle_name

class NamesTestCase(unittest.TestCase):
    """Pruebas para name_function.py."""

    def test_first_last_name(self):
        """¿Funcionan nombres como 'Ismael Serrano'?"""
        formatted_name = get_formatted_middle_name('ismael', 'serrano')
        self.assertEqual(formatted_name, 'Ismael Serrano')

if __name__ == '__main__':
    # unittest.main() # Descomentar.
    None              # Comentar
```

Esta es la salida que obtenemos al ejecutar el código:

```
E
======================================================================
ERROR: test_first_last_name (__main__.NamesTestCase.test_first_last_name)
¿Funcionan nombres como 'Ismael Serrano'?
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/n7rc/Documentos/0 - Python/11.probar_el_codigo.py", line 107, in
  test_first_last_name
    formatted_name = get_formatted_middle_name('ismael', 'serrano')
                     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
TypeError: get_formatted_middle_name() missing 1 required positional argument:
'last'

----------------------------------------------------------------------
Ran 1 test in 0.001s

FAILED (errors=1)
```

La forma de solucionar este error es hacer que el parámetro correspondiente al segundo nombre sea opcional.

Varios métodos assert.

Los métodos assert prueban si una condición que creemos que se da en un punto determinado del código se da realmente.

```
unittest.TestCase.assertEqual
unittest.TestCase.assertNotEqual
unittest.TestCase.assertTrue
unittest.TestCase.assertFalse
unittest.TestCase.assertIn
unittest.TestCase.assertNotIn
```

Esto son algunos métodos assert, pero hay muchos más. Vamos a ver cómo usar uno de ellos en la prueba de una clase.

Ejemplo: definamos una clase que ayude a administrar encuestas anónimas.

[survey.py]

```python
class AnonymousSurvey:
    """Recoge respuestas anónimas a una pregunta de una encuesta."""

    def __init__(self, question):
        """Guarda una pregunta y se prepara para guardar respuestas."""
        self.question = question
        self.responses = []

    def show_question(self):
        """Muestra la pregunta del sondeo."""
        print(self.question)

    def store_response(self, new_response):
        """Guarda una sola respuesta en la encuesta."""
        self.responses.append(new_response)

    def show_results(self):
        """Muestra todas las respuestas que se han dado."""
        print("Resultados de la encuesta:")
        for response in self.responses:
            print(f"- {response}")
```

Escribamos un programa que la use:

```
from classes.survey import AnonymousSurvey

#Define una pregunta y hace una encuesta.
question = "¿Cuál es el primer lenguaje que aprendiste?"
my_survey = AnonymousSurvey(question)

# Muestar la pregunta y guarda las respuestas a la pregunta.
my_survey.show_question()
print("Escriba 'q' para salir.\n")
while True:
    response = input("Lenguaje: ")
    if response == 'q':
        break
    my_survey.store_response(response)

#Muestra los resultados de la encuesta.

print("\nGracias a todos por participar en la encuesta.")
my_survey.show_results()
```

Probar una clase.

Vamos a escribir na prueba que verifica un aspecto del comportamiento de la clase. Escribiremos una prueba que compruebe si se ha guardado bien una respuesta simple a la encuesta. Utilizaremos el método assertIn() para comprobar que la respuesta está en la lista de respuestas después de guardarse.

```
import unittest

from classes.survey import AnonymousSurvey

class TestAnonymousSurvey(unittest.TestCase):
    """Pruebas para la clase AnonymousSurvey."""

    def test_sote_single_response(self):
        """Comprueba si la respuesta simple se guarda bien."""
        question = "¿Cuál es el lenguaje que primero aprendiste?"
        my_survey = AnonymousSurvey(question)
        my_survey.store_response('Español')
        self.assertIn('Español', my_survey.responses)

if __name__ == '__main__':
    unittest.main()
```

El primer método de la prueba comprobará si, cuando almacenamos una respuesta a la pregunta, va a parar a la lista de respuestas.

Creamos una instancia llamada my_survey.

Guardamos una respuesta simple, 'Español', con el método store_response().

Luego comprobamos si la respuesta se ha guardado bien verificando que 'Español' está en la lista my_surey.responses.

Esta es la salida que obtenemos al ejecutar el código:

```
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Cuando se ejecuta un caso de prueba, Python imprime un carácter por cada prueba unitaria cuando se completa. Una prueba que se completa imprime un punto, una que da error imprime una E y una que falla imprime una F.

APÉNDICE: INMUTABILIDAD

En Python, cada objeto se clasifica como inmutable o no. En términos de los tipos principales, los números, cadenas y tuplas son inmutables; listas y diccionarios no lo son (se pueden cambiar libremente). Entre otras cosas, la inmutabilidad se utiliza para garantizar que un objeto permanezca constante a lo largo de un programa.

```
S = 'Spam'
```

No podemos cambiar el contenido de una cadena, es un objeto inmutable. Sin embargo podemos crear una nueva variable (objeto) con el mismo nombre, asignándole el nuevo valor.

```
S[0] = 'z' # Error: TypeError: 'str' object does not support item assignment.

S = 'z' + S[1:]
print(S) # Imprime: zpam.
```

Si un tipo de objeto es inmutable, no puede cambiar su valor; Python genera un error si lo intenta. En su lugar, debe ejecutar código para crear un nuevo objeto que contenga el nuevo valor.

Por el contrario, los tipos mutables siempre se pueden cambiar con operaciones que no creen nuevos objetos. Aunque dichos objetos se pueden copiar, los cambios ayudan a la modificación directa.

Los principales tipos del núcleo en Python se desglosan de la siguiente manera:

- Inmutables: números, cadenas, tuplas, conjuntos congelados (frozenset, es un tipo de objeto, un conjunto inmutable).
- Mutables: listas, diccionarios y conjuntos (sets).

APÉNDICE: SENTENCIA 'if '__name == __main__''

Cuando un intérprete de Python lee un archivo Python, primero establece algunas variables especiales. Luego ejecuta el código desde el archivo.

Una de estas variables se llama '__name__'.

Los archivos de Python se llaman módulos y se identifican mediante la extensión de archivo .py. Un módulo puede definir funciones, clases y variables.

Cuando el intérprete ejecuta un módulo, la variable __name__ se establece como __main__ si, y solo si, el módulo que se está ejecutando es el programa principal. Pero si el código está importando el módulo, entonces la variable __name__ se establecerá en el nombre de ese módulo importado. Es decir: La variable __name__ para el archivo o módulo que se ejecuta será siempre __main__. Pero para todos los demás módulos que se importan se establecerá con el nombre del módulo.

Al leer el archivo se ejecuta todo el código de nivel superior, pero no las funciones y clases (ya que solamente se importarán).

Podemos usar un bloque if __name__ == "__name__" para permitir o evitar que se ejecuten partes del código cuando sean importados los módulos.

Para la mayoría de los propósitos prácticos, puede pensar en el bloque condicional con el que abre if __name__ == "__main__" como una forma de almacenar código que solo debería ejecutarse cuando su archivo se ejecuta como un script.

EN RESUMEN: 'if __name__ == '__main__'' le permite ejecutar código cuando el archivo se ejecuta como secuencia de comandos, pero no cuando se importa como módulo.

Veamos un ejemplo del uso de 'if __name__ == '__main__''.

[echo.py]

```
"""Al tratarse del script principal (este) el código se ejecutará"""

def echo(text: str, repetitions: int = 3):
    """Imitación del eco."""
    echoed_text = ""
    for i in range(repetitions, 0, -1):
        echoed_text += f"{text[-i:]}\n"
    return f"{echoed_text.lower()}."
```

Este código se ejecutará cuando ejecutemos este script.

```
if __name__ == '__main__':
    text = input("Introduzca cualquier entrada: ")
    print(echo(text))
```

Ahora (debajo), al importar el módulo echo.py el código no se ejecutará. Sin embargo podemos usar la función echo() tal y como muestra la sentencia print().

```
from modules.echo import echo # Aquí el código no se ejecuta, solo se importa.

print(echo("Holaaaa"))
```

APÉNDICE: MÓDULOS Y PAQUETES

Los módulos en Python son simplemente archivos Python con la extensión .py, que implementan un conjunto de funciones. Los módulos son importados desde otros módulos usando el comando import.

La primera vez que se carga un módulo dentro de un script Python en ejecución, éste se inicializa ejecutando el código del módulo una vez. Si otro módulo en tu código importa el mismo módulo de nuevo, éste no se cargará una segunda vez, solo son inicializadas una vez.

En general, los programas de Python se componen de varios archivos de módulos, vinculados entre sí por declaraciones de importación. Cada archivo de módulo es un paquete autónomo de variables, es decir, un espacio de nombres (namespace). Un archivo de módulo no puede ver los nombres definidos en otro archivo a menos que importe explícitamente ese otro archivo, por lo que los módulos sirven para minimizar las colisiones de nombres en su código: dado que cada archivo es un espacio de nombres independiente, los nombres de un archivo no pueden chocar con los de otro, incluso si se escriben de la misma manera.

Importación de atributos de un módulo.

Importamos el módulo modules.module

[module.py]

```
"""
Define atributos que serán exportados a otros archivos.
En el módulo se asignan tres variables, y estas, estarán disponibles como
atributos para el exterior.
"""
a = 'alberto' # Se definen tres atributos (exportables a otros ficheros).
b = 'mónica'
c = 'sílvia'

print(f"Primera impresión desde 'module.py' {a}, {b}, {c}")

if __name__ == '__main__':

    # Esta sentencia solo se ejecutará cuando se ejecute este script como script
    # principal. Si es importado por otro script no se ejecutará.

    print(f"Segunda impresión desde 'module.py' {a}, {b}, {c}")
```

```
from modules import module

print(f"Impresión de atributos de 'module.py':{module.a}, {module.b},
      {module.c}")
```

Han ocurrido varias cosas aquí:

Podemos ver como en print podemos llamar a los atributos del módulo, por ejemplo:

```
'{module.a}'.
```

Como ya hemos comentado, en módulo module.py contiene la sentencia de Python if __name__ == '__main__', con la cual, el código que forme parte de ella no se

ejecutará en la importación, pero sí lo hará cualquier otro código que no forme parte del bloque de esa sentencia. Este cualquier otro código, por ejemplo, la primera llamada a print desde module.py, se ejecutará solo una vez con independencia de cuántas veces importemos el módulo.

La ejecución de este archivo tiene como resultado el siguiente:

Primera impresión desde 'module.py' alberto, mónica, sílvia

Impresión de atributos de 'module.py':alberto, mónica, sílvia

También podemos importar los atributos del módulo por separado, de esta forma estaremos copiando las variables del módulo module.py a nuestro script. Lo vemos en la siguiente línea:

```
from modules.module import a, b, c

print(f"'{c.title()}' Es una variable importada de 'module.py'")
```

La función dir().

La función, incorporada, de la biblioteca estándar, dir(), es muy útil para obtener una lista de los nombres disponibles dentro de un módulo. Cuando la función dir se llama con el nombre de un módulo importado entre paréntesis, como aquí, devuelve todos los atributos dentro de ese módulo.

```
print(dir(module))
```

Devuelve:

```
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__',
'__package__', '__spec__', 'a', 'b', 'c']
```

```
S = 'cadena'
print (dir(S))
```

Devuelve:

```
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__',
'__getnewargs__',
'__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__',
'__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__',
'__sizeof__',
'__str__', '__subclasshook__', 'capitalize', 'casefold', 'center', 'count',
'encode',
'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum',
'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier', 'islower',
'isnumeric',
'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower',
'lstrip',
'maketrans', 'partition', 'removeprefix', 'removesuffix', 'replace', 'rfind',
'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines',
'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

dir() devuelve una lista de todos los atributos disponibles para un objeto dado. Dado que los métodos son "atributos de función" aparecerán en esta lista. De este modo podemos ver la lista de métodos par aun objeto dado. En esta caso una cadena (string). Los nombres sin guiones bajos en esta lista son los métodos invocables en el objeto cadena.

La función help().

Cuando encontramos en un módulo la función que queremos usar, mediante el comando dir(), en el intérprete de python podemos obtener más información sobre la misma con el comando help().

```
# >>> import subprocess
# >>> help(subprocess.cal)
```

```
from subprocess import call
print(help(call))
```

Imprime:

```
Help on function call in module subprocess:

call(*popenargs, timeout=None, **kwargs)
    Run command with arguments.  Wait for command to complete or
    timeout, then return the returncode attribute.

    The arguments are the same as for the Popen constructor.  Example:

    retcode = call(["ls", "-l"])
```

dir() y help() son la primera línea de documentación en Python.

Paquetes.

Los paquetes son espacios de nombre que contienen múltiples módulos. Son simplemente directorios. Cada paquete de Python es un directorio que debe contener un archivo especial llamado '__init__.py'. Este archivo puede estar vacío, e indica que el contenido del directorio es un paquete Python; de este modo puede ser importado de la misma forma en la que se importa un módulo.

Por ejemplo podemos crear el directorio modules y crear el módulo module en él (añadiendo el archivo module.py), ahora, si además incorporamos a modules el archivo __init__.py, este directorio se convertirá en un paquete para Python.

Podemos importar el módulo module(.py) de dos formas:

```
import modules.module
```

o

```
from modules import module
```

Con el primer método, debemos usar el prefijo 'modules' cuando accedamos a los atributos o métodos del módulo 'module'. Con el segundo método no necesitamos usar el prefijo, porque estamos importando todo el módulo al espacio de nombres del modulo desde el que importamos.

El archivo __init__.py puede, además, decidir qué módulos exporta el paquete como API, mientras mantiene otros módulos internamente. Esto lo podemos conseguir sobrescribiendo la variable __all__ en el archivo __init__.py

```
Paquete (directorio) modules.
__init__.py
__all__ == ['module']
```

En las líneas anteriores estamos diciendo que el paquete 'modules' solo exporte como API el módulo 'module'.

import vs from.

La instrucción from, en cierto sentido, anula el propósito de separación de los espacios de nombres de los módulos; dado que copia las variables de un archivo a otro. Esto puede causar que si hay variables con el mismo nombre en el archivo importado que en el archivo desde el que hacemos la importación, estas se sobrescribirán y no habrá aviso de tal cosa. Esto colapsa los espacios de nombre, al menos, en término de las variables copiadas. Por este motivo algunos recomiendan usar import en lugar de from. Pero rara vez se da este problema, además de que el uso de from implica escribir menos.

APÉNDICE: ESCRITURA DINÁMICA (TIPADO DINÁMICO)

La idea más fundamental en la programación de Python y la base de gran parte de la concisión y la flexibilidad del lenguaje, es sin duda la escritura dinámica (tipado dinámico) y el polimorfismo que produce.

Como hemos visto, en Python no declaramos los tipos específicos de los objetos que usan nuestros scripts. De hecho, los programas ni siquiera deberían preocuparse por tipos específicos; a cambio, son naturalmente aplicables en más contextos de los que a veces, incluso podemos planificar con anticipación. Debido a que la escritura dinámica es la raíz de esta flexibilidad, echemos un breve vistazo al modelo aquí.

Actualmente, los dos enfoques más comunes para la escritura de variables son la escritura estática y la escritura dinámica. Cuando se trata de tipos de manejo, ambos enfoques ofrecen sus propios conjuntos de beneficios y dificultades. Sin embargo, en esencia, la elección se reduce a lo que más necesita: estabilidad operativa y código limpio, o agilidad y flexibilidad de desarrollo.

En lenguajes de tipado dinámico como Python, JavaScript, PHP y Perl, la verificación de tipos ocurre en tiempo de ejecución. En lugar de detener las operaciones, el compilador ignorará cosas como argumentos de tipos no válidos o datos no coincidentes; en su lugar, la verificación de tipos se realizará con regularidad durante el tiempo de ejecución. Si el verificador de tipo detecta un error, alertará al desarrollador y le dará la oportunidad de corregir el código antes de que el programa falle.

Un lenguaje de programación tiene un sistema de tipos (esta es una mejor forma de ponerlo en español) dinámico cuando el tipo de dato de una variable puede cambiar en tiempo de ejecución. Python efectivamente es, entonces, un lenguaje de tipado dinámico, pues una variable puede comenzar teniendo un tipo de dato y cambiar en cualquier momento a otro tipo de dato. Por ejemplo:

```
a = 5
print(a)
a = "Hola mundo"
print(a)
```

Si Python fuera un lenguaje de tipado estático, la segunda asignación de 'a' con una cadena debería arrojar un error, pues su tipo de dato es entero.

Python es de tipado dinámico, como hemos dicho, y fuertemente tipado. Por ejemplo, imagina que un desarrollador de Python declara una variable basada en números, pero la vincula a un método diseñado para manejar variables de texto (ten en cuenta que Python no requiere que declares tipos de variables antes de asignarles un valor):

```
var = 10
print (len(var))    # Error: TypeError: object of type 'int' has no len()
```

Dado que el método len() no puede aceptar una variable numérica como argumento, esto producirá un error. Sin embargo, a pesar del error tipográfico, el compilador no se preocupa por el tipo de variable y aún permitirá que el programa se ejecute. Una vez que se detecta la falta de coincidencia durante el tiempo de ejecución, el verificador de tipo pausará la ejecución, arrojará un error y presentará la posibilidad de ajustar la variable o la función de llamada según sea necesario.

En conclusión, Python es un lenguaje de tipado dinámico y fuerte, con la posibilidad de adquirir un sistema de tipado estático vía las anotaciones de tipos como las siguientes:

```
a: int = 5              # Tipado estático.
b: str = "Hola mundo"

print(a)
print(b)
```

Cuando escribimos 'a = 5' en una sesión interactiva o en un archivo de programa, por ejemplo, ¿cómo sabe Python que 'a' debe representar un número entero? De hecho, ¿cómo sabe Python qué es 'a' ?

En Python, los tipos se determinan automáticamente en tiempo de ejecución, no en respuesta a declaraciones en su código. Esto significa que nunca declara variables con anticipación (un concepto que quizás sea más sencillo de comprender si tiene en cuenta que todo se reduce a variables, objetos y los vínculos entre ellos). Este es un concepto diferente al abordado por otros lenguajes, como por ejemplo C, con tipado estático.

Cuando ejecuta una declaración de asignación como 'a = 5' en Python, funciona incluso si nunca le ha dicho a Python que use el nombre 'a' como una variable, o que 'a' deba representar un objeto de tipo entero.

Creación de variables.

Una variable (es decir, un nombre), como 'a', se crea cuando su código le asigna un valor por primera vez. Las asignaciones futuras cambian el valor del nombre ya creado. Técnicamente, Python detecta algunos nombres antes de que se ejecute su código, pero puede pensar en ello como si las asignaciones iniciales crearan variables.

Tipos de variables.

Una variable nunca tiene ningún tipo de información o restricciones asociadas con ella. La noción de tipo vive con objetos, no con nombres. Las variables son de naturaleza genérica; siempre se refieren simplemente a un objeto particular en un momento particular.

Uso variable.

Cuando aparece una variable en una expresión, se reemplaza inmediatamente con el objeto al que se refiere actualmente, cualquiera que sea. Además, todas las variables deben asignarse explícitamente antes de que puedan usarse; hacer referencia a variables no asignadas da como resultado errores.

En resumen, las variables se crean cuando se asignan, pueden hacer referencia a cualquier tipo de objeto y deben asignarse antes de que se haga referencia a ellas.

Por ejemplo, cuando declaramos:

```
a = 5
```

Python realizará tres pasos distintos para llevar a cabo la solicitud. Estos pasos reflejan el funcionamiento de todas las asignaciones en el lenguaje Python:

1. Crea un objeto para representar el valor 3.
2. Crea la variable 'a', si aún no existe.
3. Vincula la variable 'a' al nuevo objeto 3.

Estos enlaces de variables a objetos se denominan referencias en Python, es decir, una referencia es un tipo de asociación, implementada como un puntero en la memoria.

A un programador de C le resultaría extraño este trozo de código:

```
a = 5               # 'a' es un entero.
a = "Hola, mundo"   # Ahora 'a' es una cadena.
a = 1.2345          # Ahora 'a' es un número en punto flotante.
```

Los nombres ('a') no tienen tipos; como se indicó anteriormente, los tipos viven con objetos, no con nombres. En la lista anterior, simplemente hemos cambiado 'a' para hacer referencia a diferentes objetos. Como las variables no tienen tipo,en realidad no hemos cambiado el tipo de la variable 'a'; simplemente hemos hecho que la variable haga referencia a un tipo diferente de objeto. De hecho, nuevamente, todo lo que podemos decir acerca de una variable en Python es que hace referencia a un objeto particular en un momento particular.

Los objetos, por otro lado, saben de qué tipo son: cada objeto contiene un campo de encabezado que etiqueta el objeto con su tipo. Debido a que los objetos conocen sus tipos, las variables no tienen que hacerlo. En resumen, los tipos están asociados con objetos en Python, no con variables.

En el nivel más práctico, el tipado dinámico significa que hay menos código para escribir. Sin embargo, igual de importante es que el tipado dinámico es también la raíz del polimorfismo de Python, un concepto que presentamos más tarde.

Polimorfismo.

El término polimorfismo se refiere, esencialmente, a que el significado de una operación depende de los objetos que se están operando. Debido a que es un lenguaje tipado dinámicamente, el polimorfismo corre rampante en Python. De hecho, cada operación es una operación polimórfica en Python: impresión, indexación, el operador '*' y mucho más. Por ejemplo:

```
print(3 * 4)         # Imprime: 12.
print('¡No!' * 4)    # El mismo operador imprime: ¡No!¡No!¡No!¡No!.
```

Es decir, es un término usado en Python para referirse a un nombre de función genérico que puede usarse para varios tipos."""

Polimorfismo, clases y 'Duck typing'.

Si camina como un pato y habla como un pato, entonces tiene que ser un pato. Esta frase es el origen del concepto 'duck typing'. Es un símil en el que los patos son objetos y hablar/andar métodos. Se traduciría así como que si un objeto tiene los métodos que nos interesan, nos basta, siendo su tipo irrelevante.

Dicho coloquialmente, a Python le dan igual los tipos de los objetos, lo que le importan son los métodos.

Creamos la clase Pato y el método hablar().

```
class Pato:
    def hablar(self):
        print("¡Cua!, ¡Cua!")

p = Pato()
p.hablar()          # Imprime: ¡Cua!, ¡Cua!.
```

Ahora creamos una función: llama_hablar(), que llama al método hablar() del objeto que se le pase.

```
class Pato:
    def hablar(self):
        print("¡Cua!, ¡Cua!")

def llama_hablar(x):
        x.hablar()

p = Pato()
llama_hablar(p)     # Imprime: ¡Cua!, ¡Cua!.
```

Cuando Python entra en la función y evalúa x.hablar(), le da igual el tipo al que pertenezca x siempre y cuando tenga el método hablar(). Esto es lo que se llama duck typing. Comprobemos esto:

Creamos otra clase diferente a Pato, la clase Gato:

```
class Gato:
    def hablar(self):
        print("¡Miau!, ¡Miau!")

g = Gato()
# La función llama_hablar() funciona con objetos diferentes.
llama_hablar(p)     # Imprime: ¡Cua!, ¡Cua!
llama_hablar(g)     # Imprime: ¡Miau!, ¡Miau!
```

Python es un lenguaje que soporta el duck typing, lo que hace que el tipo de los objetos no sea tan relevante, siendo más importante lo que pueden hacer (sus métodos).

El duck typing, como hemos visto, está en todos lados, desde la función len() hasta el uso del operador *.