Una nota sobre bloques de código y sintaxis

Al escribir código, es muy importante usar la sintaxis correcta. Incluso un pequeño error tipográfico, como un paréntesis que falta o una coma adicional, puede provocar un error de sintaxis y el código no se ejecutará en absoluto. ¡Ay! Si su código da como resultado un error o una excepción, preste mucha atención a la sintaxis y tenga cuidado con los errores menores.

Si su sintaxis es correcta, pero la secuencia de comandos tiene un comportamiento o una salida inesperados, esto puede deberse a un problema semántico. Recuerde que la sintaxis son las reglas de cómo se construye el código, mientras que la semántica es el efecto general que tiene el código. Es posible tener un código sintácticamente correcto que se ejecute correctamente, pero no haga lo que queremos que haga.

Cuando trabaje con los bloques de código en los ejercicios de este curso, tenga en cuenta los errores de sintaxis, junto con el resultado general de su código. ¡El hecho de que haya corregido un error de sintaxis no significa que el código tendrá el efecto deseado cuando se ejecute! Una vez que haya corregido un error en su código, no olvide enviarlo para que se verifique su trabajo.

# Más sobre Python

## **Más sobre Python**

### **Usando Python por su cuenta**

La mejor manera de aprender cualquier lenguaje de programación es practicarlo por tu cuenta tanto como puedas. Si tiene Python instalado en su computadora, puede ejecutar el intérprete ejecutando el comando python3 (o simplemente python en Windows), y puede cerrarlo escribiendo exit () o Ctrl-D.

Si aún no tiene Python instalado en su máquina, está bien. Explicaremos cómo instalarlo en un próximo curso.

Mientras tanto, todavía puede practicar utilizando uno de los muchos intérpretes o teclados de código de Python disponibles en línea. No hay mucha diferencia entre un intérprete y un teclado. Un intérprete es más interactivo que un teclado, pero ambos le permiten ejecutar código y ver los resultados.

A continuación, encontrará enlaces a algunos de los intérpretes y teclados en línea más populares. Pruébalos para encontrar tu favorito.

* <https://www.python.org/shell/>
* <https://www.onlinegdb.com/online_python_interpreter>
* <https://repl.it/languages/python3>
* <https://www.tutorialspoint.com/execute_python3_online.php>
* <https://rextester.com/l/python3_online_compiler>
* <https://trinket.io/python3>

### **Recursos adicionales de Python**

Si bien este curso le brindará información sobre cómo funciona Python y cómo escribir scripts en Python, es probable que desee obtener más información sobre partes específicas del lenguaje. A continuación, se muestran algunas formas excelentes de ayudarlo a encontrar información adicional:

* Lea la [documentación oficial de Python](https://docs.python.org/3/) .
* Busque respuestas o haga una pregunta en [Stack Overflow](https://stackoverflow.com/) .
* Suscríbase a la lista de correo de [tutores de](https://mail.python.org/mailman/listinfo/tutor) Python , donde puede hacer preguntas y colaborar con otros estudiantes de Python.
* Suscríbase a la lista de correo de [Python](https://mail.python.org/mailman/listinfo/python-announce-list) -nounce para leer sobre las últimas actualizaciones en el idioma.

### **Historia de Python y estado actual**

Python fue lanzado hace casi 30 años y tiene una rica historia. Puede leer más sobre esto en la página de Wikipedia [de History of Python](https://en.wikipedia.org/wiki/History_of_Python) o en la sección sobre el [historial del software](https://docs.python.org/3.0/license.html) de la documentación oficial de Python.

Python ha sido llamado recientemente el lenguaje de programación de más rápido crecimiento. Si está interesado en saber por qué es esto y cómo se mide, puede encontrar más información en estos artículos:

* [El increíble crecimiento de Python](https://stackoverflow.blog/2017/09/06/incredible-growth-python/) (desbordamiento de pila)
* [¿Por qué Python está creciendo tan](https://www.netguru.com/blog/why-python-is-growing-so-quickly-future-trends) rápido? [Tendencias futuras](https://www.netguru.com/blog/why-python-is-growing-so-quickly-future-trends) (Netguru)
* [En cifras: tendencias de la comunidad de Python en 2017/2018](https://opensource.com/article/18/5/numbers-python-community-trends) (Opensource.com)
* [Resultados de la encuesta para desarrolladores 2018](https://insights.stackoverflow.com/survey/2018#technology) (desbordamiento de pila)

# Hoja de referencia de los primeros conceptos de programación

## **Hoja de referencia de los primeros conceptos de programación**

### **Funciones y palabras clave**

Las funciones y las palabras clave son los componentes básicos de la sintaxis de un idioma.

Las funciones son piezas de código que realizan una unidad de trabajo. En los ejemplos que hemos visto hasta ahora, solo hemos encontrado la función print (), que imprime un mensaje en la pantalla. Aprenderemos sobre muchas otras funciones en lecciones posteriores pero, si tiene demasiada curiosidad como para esperar hasta entonces, puede descubrir todas las funciones disponibles [aquí](https://docs.python.org/3/library/functions.html) .

Las palabras clave son palabras reservadas que se utilizan para construir instrucciones. Nos encontramos brevemente para y en nuestro primer ejemplo de Python, y usaremos un montón de otras palabras clave a medida que avanzamos en el curso. Como referencia, estas son todas las palabras clave reservadas:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| False | class | finally | is | return |
| None | continue | for | lambda | try |
| True | def | from | nonlocal | while |
| and | del | global | not | with |
| as | elif | if | or | yield |
| assert | else | import | pass |  |
| break | except | in | raise |  |

No necesita aprender esta lista; profundizaremos en cada palabra clave a medida que las encontremos. Mientras tanto, puede ver ejemplos de uso de palabras clave [aquí](https://www.programiz.com/python-programming/keyword-list) .

### **Operadores aritméticos**

Python puede operar con números usando los operadores matemáticos habituales y también con algunos operadores especiales. Estos son todos (exploraremos los dos últimos en videos posteriores).

* **a + b** = suma a y b
* **a - b** = Resta b de a
* **a \* b** = Multiplica ay b
* **a / b** = Divide a entre b
* **a \*\* b** = Eleva a a la potencia de b. Para valores no enteros de b, esto se convierte en una raíz (es decir, a \*\* (1/2) es la raíz cuadrada de a)
* **a // b** = La parte entera de la división entera de a por b
* **a% b** = La parte restante de la división entera de a por b

# Resumen de tipos de datos

En Python, el texto entre comillas, ya sea comillas simples o dobles, es un tipo de datos de cadena. Un entero es un número entero, sin fracción, mientras que un flotante es un número real que puede contener una parte fraccionaria. Por ejemplo, 1, 7, 342 son todos números enteros, mientras que 5.3, 3.14159 y 6.0 son todos flotantes. Al intentar mezclar tipos de datos incompatibles, puede encontrar un **TypeError** . Siempre puede verificar el tipo de datos de algo usando la función type () .

# Conversión implícita vs explícita

Como vimos anteriormente en el video, algunos tipos de datos se pueden mezclar y combinar debido a la conversión implícita. La conversión implícita es donde el intérprete nos ayuda y convierte automáticamente un tipo de datos en otro, sin tener que decirle explícitamente que lo haga.

Por el contrario, la conversión explícita es donde convertimos manualmente de un tipo de datos a otro llamando a la función relevante para el tipo de datos al que queremos convertir. Usamos esto en nuestro ejemplo de video cuando queríamos imprimir un número junto con algún texto. Antes de que pudiéramos hacer eso, necesitábamos llamar a la función str () para convertir el número en una cadena. Una vez que el número se convirtió explícitamente en una cadena, podríamos unirlo con el resto de nuestra cadena textual e imprimir el resultado.

# Resumen de la definición de funciones

Vimos algunos ejemplos de funciones integradas en Python, pero poder definir sus propias funciones es increíblemente poderoso. Comenzamos una definición de función con la palabra clave def, seguida del nombre que queremos darle a nuestra función. Después del nombre, tenemos los parámetros, también llamados argumentos, para la función entre paréntesis. Una función no puede tener parámetros o puede tener varios parámetros. Los parámetros nos permiten llamar a una función y pasarle datos, estando los datos disponibles dentro de la función como variables con el mismo nombre que los parámetros. Por último, colocamos dos puntos al final de la línea.

Después del colon, comienza el cuerpo funcional. Es importante tener en cuenta que en Python el cuerpo de la función está delimitado por sangría. Esto significa que todo el código sangrado a la derecha después de la definición de una función es parte del cuerpo de la función. La primera línea que ya no tiene sangría es el límite del cuerpo de la función. Depende de usted cuántos espacios use al sangrar, solo asegúrese de ser consistente. Entonces, si elige sangrar con cuatro espacios, debe usar cuatro espacios en todas partes en su código.

# Devolución de valores mediante funciones

A veces no queremos que una función simplemente se ejecute y finalice. Es posible que deseemos que una función manipule los datos que le pasamos y luego nos devuelva el resultado. Aquí es donde el concepto de valores devueltos resulta útil. Usamos la palabra clave return en una función, que le dice a la función que devuelva los datos. Cuando llamamos a la función, podemos almacenar el valor devuelto en una variable. Los valores de retorno permiten que nuestras funciones sean más flexibles y potentes, por lo que se pueden reutilizar y llamar varias veces.

Las funciones pueden incluso devolver varios valores. ¡No olvide almacenar todos los valores devueltos en variables! También puede hacer que una función no devuelva nada, en cuyo caso la función simplemente sale.

Operadores de comparación

En Python, podemos usar operadores de comparación para comparar valores. Cuando se realiza una comparación, Python devuelve un resultado booleano, o simplemente un Verdadero o Falso.

* Para comprobar si dos valores son iguales, podemos usar el operador de igualdad: **==**
* Para comprobar si dos valores no son iguales, podemos usar el operador no es igual:! **=**

También podemos comprobar si los valores son mayores o menores entre sí usando **>** y **<** . Si intenta comparar tipos de datos que no son compatibles, como comprobar si una cadena es mayor que un número entero, Python arrojará un **TypeError** .

Podemos hacer comparaciones muy complejas uniendo declaraciones usando operadores lógicos con nuestros operadores de comparación. Estos operadores lógicos son **y** , **o** y **no** . Cuando se usa el operador **y** , ambos lados de la declaración que se evalúa deben ser verdaderos para que toda la declaración sea verdadera. Cuando se usa el operador **o** , si cualquiera de los lados de la comparación es verdadero, entonces toda la afirmación es verdadera. Por último, el operador **not** simplemente invierte el valor de la declaración que le sigue inmediatamente. Entonces, si una declaración se evalúa como Verdadera y colocamos el operador **not** delante de ella, se volverá Falsa.

# if Resumen de declaraciones

Podemos usar el concepto de **ramificación** para que nuestro código altere su secuencia de ejecución dependiendo de los valores de las variables. Podemos usar una declaración if para evaluar una comparación. Comenzamos con la palabra clave if , seguida de nuestra comparación. Terminamos la línea con dos puntos. El cuerpo de la instrucción if se sangra a la derecha. Si la comparación es **Verdadera** , se ejecuta el código dentro del cuerpo if . Si la comparación se evalúa como **Falsa** , el bloque de código se omite y no se ejecutará.

# otras declaraciones y el operador de módulo

Acabamos de cubrir la declaración if , que ejecuta código si una evaluación es verdadera y omite el código si es falsa. Pero, ¿y si quisiéramos que el código hiciera algo diferente si la evaluación es falsa? Podemos hacer esto usando la instrucción else . La instrucción else sigue a un bloque if y se compone de la palabra clave else seguida de dos puntos. El cuerpo de la instrucción else está sangrado a la derecha y se espera si no se ejecuta la instrucción if anterior .

También tocamos el operador de módulo, que está representado por el signo de porcentaje: **%** . Este operador realiza una división de enteros, pero solo devuelve el resto de esta operación de división. Si dividimos 5 entre 2, el cociente es 2 y el resto es 1. Dos 2 pueden entrar en 5, dejando 1 sobrante. Entonces, 5% 2 devolvería 1. Dividir 10 entre 5 nos daría un cociente de 2 sin resto, ya que 5 puede entrar en 10 dos veces sin que quede nada. En este caso, 10% 2 devolvería 0, ya que no hay resto.

# Bifurcaciones más complejas con declaraciones elif

Partiendo de los bloques if y else , que nos permiten ramificar nuestro código dependiendo de la evaluación de una declaración, la declaración elif nos permite incluso más comparaciones para realizar ramificaciones más complejas. Muy similar a las declaraciones if , una declaración elif comienza con la palabra clave elif , seguida de una comparación para ser evaluada. A esto le siguen dos puntos y luego el bloque de código de la línea siguiente, con sangría a la derecha. Una declaración elif debe seguir a una declaración if y solo se evaluará si la declaración if se evaluó como falsa. Puede incluir varios elifdeclaraciones para construir ramificaciones complejas en su código para hacer todo tipo de cosas poderosas.

# Hoja de referencia de condicionales

## **Hoja de referencia de condicionales**

En videos anteriores, echamos un vistazo a algunos de los operadores de Python integrados que nos permiten comparar valores y algunos operadores lógicos que podemos usar para combinar valores. También aprendimos cómo usar operadores en bloques if-else-elif.

Es mucho que aprender pero, con la práctica, es más fácil recordarlo todo. Mientras tanto, esta práctica hoja de trucos le brinda toda la información que necesita de un vistazo.

### **Operadores de comparación**

* a == b: a es igual ab
* a! = b: a es diferente de b
* a <b: a es más pequeño que b
* a <= b: a es menor o igual ab
* a> b: a es mayor que b
* a> = b: a es mayor o igual ab

### **Operadores lógicos**

* ayb: Verdadero si tanto a como b son Verdaderos. Falso de lo contrario.
* aob: Verdadero si aob o ambos son Verdaderos. Falso si ambos son falsos.
* no a: Verdadero si a es Falso, Falso si a es Verdadero.

**Bloques de ramificación**

En Python, ramificamos nuestro código usando if, else y elif. Esta es la sintaxis de ramificación:

if condition1:

    if-block

elif condition2:

    elif-block

else:

    else-block

Recuerde: el bloque if se ejecutará si condition1 es True. El bloque elif se ejecutará si condition1 es False y condition2 es True. El bloque else se ejecutará cuando todas las condiciones especificadas sean falsas.

Anatomía de un bucle while

Un *tiempo* de bucle se ejecutará continuamente código en función del valor de una condición. Comienza con la palabra clave *while,* seguida de una comparación para ser evaluada, luego dos puntos. En la siguiente línea está el bloque de código que se ejecutará, sangrado a la derecha. Similar a una declaración *if* , el código en el cuerpo solo se ejecutará si la comparación se evalúa como verdadera. Lo que diferencia a un *tiempo* de bucle aparte, sin embargo, es que este bloque de código mantendrá la ejecución, siempre y cuando el estado de la evaluación es cierto. Una vez que la declaración ya no es cierta, el ciclo sale y se ejecuta la siguiente línea de código.

Errores comunes con la inicialización variable

Deberá estar atento a un error común: olvidarse de inicializar las variables. Si intenta utilizar una variable sin inicializarla primero, se encontrará con un **NameError** . Este es el intérprete de Python que detecta el error y le dice que está usando una variable indefinida. La solución es bastante simple: inicialice la variable asignándole un valor antes de usarla.

Otro error común a tener en cuenta y que puede ser un poco más complicado de detectar es olvidarse de inicializar las variables con el valor correcto. Si usa una variable anteriormente en su código y luego la reutiliza más tarde en un ciclo sin establecer primero el valor en algo que desea, su código puede terminar haciendo algo que no esperaba. ¡No olvide inicializar sus variables antes de usarlas!

Bucles infinitos y bloques de código

Otro error fácil que puede ocurrir cuando se utilizan bucles es introducir un bucle infinito. Un bucle infinito significa que el bloque de código en el bucle continuará ejecutándose y nunca se detendrá. Esto puede suceder cuando la condición que está siendo evaluado en un *tiempo* bucle no cambia. Preste mucha atención a sus variables y los posibles valores que pueden tomar. Piense en valores inesperados, como cero.

En los bloques de código de Coursera, es posible que vea un mensaje de error que dice "La evaluación tardó más de 5 segundos en completarse". Esto significa que el código encontró un bucle infinito y se agotó el tiempo de espera después de 5 segundos. Debería mirar más de cerca el código y las variables para detectar dónde está el bucle infinito.

Para resumen de bucles

*Los* bucles *for le* permiten iterar sobre una secuencia de valores. Tomemos el ejemplo del comienzo del video:

para x en el rango (5):

  imprimir (x)

Al igual que en *caso de* declaraciones y *mientras* bucles, *de* bucles comienzan con la palabra clave ***para los*** dos puntos al final de la línea. Al igual que en la definición de funciones, *mientras que* los bucles y *si* las declaraciones, el cuerpo de la *de* bucle comienza en la línea siguiente y se sangra a la derecha. Pero, ¿qué pasa con las cosas entre la palabra clave *for* y los dos puntos? En nuestro ejemplo, estamos usando la función *range ()* para crear una secuencia de números sobre la que nuestro bucle *for* puede iterar. En este caso, nuestra variable **x** apunta al elemento actual en la secuencia como *para*bucle itera sobre la secuencia de números. Tenga en cuenta que en Python y en muchos lenguajes de programación, un rango de números comenzará en 0 y la lista de números generada será uno menos que el valor proporcionado. Entonces, *range (5)* generará una secuencia de números del 0 al 4, para un total de 5 números.

Al reunir todo esto, la función range (5) creará una secuencia de números del 0 al 4. Nuestro ciclo *for* iterará sobre esta secuencia de números, uno a la vez, haciendo que los números sean accesibles a través de la variable **x** y el código dentro nuestro cuerpo de bucle se ejecutará para cada iteración a través de la secuencia. Entonces, para el primer ciclo, **x** contendrá 0, el siguiente ciclo, 1, y así sucesivamente hasta llegar a 4. Una vez que llegue el final de la secuencia, el ciclo saldrá y el código continuará.

El poder de los bucles *for* proviene del hecho de que puede iterar sobre una secuencia de cualquier tipo de datos, no solo un rango de números. Puede usar bucles *for* para iterar sobre una lista de cadenas, como nombres de usuario o líneas en un archivo.

¿No está seguro de si se debe utilizar una de bucle o un tiempo de bucle? Recuerde que un *tiempo* de bucle es ideal para realizar una acción una y otra vez hasta que una condición ha cambiado. Un bucle *for* funciona bien cuando desea iterar sobre una secuencia de elementos.

# Una mirada más cercana a la función Range ()

Anteriormente habíamos usado la función range () pasándole un solo parámetro, y generó una secuencia de números de 0 a uno menos de lo que especificamos. Pero la función range () puede hacer mucho más que eso. Podemos pasar dos parámetros: el primero especifica nuestro punto de partida, el segundo especifica el punto final. No olvide que la secuencia generada no contendrá el último elemento; detendrá uno antes del parámetro especificado.

La función range () también puede tomar un tercer parámetro. Este tercer parámetro le permite modificar el tamaño de cada paso. Entonces, en lugar de crear una secuencia de números incrementada en 1, puede generar una secuencia de números que se incrementa en 5.

Para recapitular rápidamente la función range () al pasar uno, dos o tres parámetros:

* Un parámetro creará una secuencia, uno por uno, de cero a uno menos que el parámetro.
* Dos parámetros crearán una secuencia, uno por uno, desde el primer parámetro hasta uno menos que el segundo.
* Tres parámetros crearán una secuencia comenzando con el primer parámetro y deteniéndose antes del segundo parámetro, pero esta vez aumentando cada paso por el tercer parámetro.

# Hoja de referencia de bucles

## **Hoja de referencia de bucles**

Consulte a continuación para ver un resumen de la sintaxis de los bucles while y for.

### **Mientras bucles**

Un ciclo while ejecuta el cuerpo del ciclo mientras la condición permanece como Verdadero.

Sintaxis:

while condition:

    body

¡Cosas a tener en cuenta!

* **Error al inicializar las variables.** Asegúrese de que todas las variables utilizadas en la condición del bucle se inicialicen antes del bucle.
* **Bucles infinitos no deseados.** Asegúrese de que el cuerpo del bucle modifique las variables utilizadas en la condición, de modo que el bucle termine finalmente **para todos los valores posibles de las variables.**

Uso típico:

Mientras que los bucles se utilizan principalmente cuando hay un número desconocido de operaciones que realizar y es necesario comprobar una condición en cada iteración.

### **Para bucles**

Un bucle for itera sobre una secuencia de elementos, ejecutando el cuerpo del bucle para cada elemento de la secuencia.

Sintaxis:

for variable in sequence

    body

La función range ():

range () genera una secuencia de números enteros. Puede tomar uno, dos o tres parámetros:

* rango (n): 0, 1, 2, ... n-1
* range(x,y): x, x+1, x+2, ... y-1
* rango (p, q, r): p, p + r, p + 2r, p + 3r, ... q-1 (si es un incremento válido)

**Errores comunes:**

* **Olvidar que el límite superior de un rango () no está incluido.**
* **Iterando sobre no secuencias.** Los números enteros no son iterables. Las cadenas son iterables letra por letra, pero es posible que eso no sea lo que desea.

Uso típico:

Los bucles for se utilizan principalmente cuando hay una secuencia predefinida o un rango de números para iterar.

### **Romper y continuar**

Puede interrumpir los bucles while y for utilizando la palabra clave break. Normalmente hacemos esto para interrumpir un ciclo debido a una condición separada.

Puede usar la palabra clave continue para omitir la iteración actual y continuar con la siguiente. Esto se usa generalmente para avanzar cuando algunos de los elementos de la secuencia no son relevantes.

Si desea obtener más información, consulte esta [página wiki en bucles](https://wiki.python.org/moin/ForLoop) .

# Fuentes de recursividad adicionales

## **Fuentes de recursividad adicionales**

En los videos anteriores, visitamos los conceptos básicos de funciones recursivas.

Una función recursiva debe incluir un caso recursivo y un caso base. El caso recursivo vuelve a llamar a la función, con un valor diferente. El caso base devuelve un valor sin llamar a la misma función.

Una función recursiva generalmente tendrá esta estructura:

def recursive\_function(parameters):

    if base\_case\_condition(parameters):

        return base\_case\_value

    recursive\_function(modified\_parameters)

Para obtener más información sobre la recursividad, consulte estos recursos:

* [Página de recurrencia de Wikipedia](https://en.wikipedia.org/wiki/Recursion)
* Vea lo que sucede cuando [busca recurrencia en Google](https://www.google.com/search?q=recursion)

Indexación y corte de cadenas

La indexación de cadenas le permite acceder a caracteres individuales en una cadena. Puede hacer esto usando corchetes y la ubicación, o índice, del carácter al que desea acceder. Es importante recordar que Python comienza los índices en 0. Entonces, para acceder al primer carácter de una cadena, debe usar el índice [0]. Si intenta acceder a un índice que es más grande que la longitud de su cadena, obtendrá un **IndexError** . ¡Esto se debe a que estás intentando acceder a algo que no existe! También puede acceder a los índices desde el final de la cadena hacia el inicio de la cadena mediante el uso de valores negativos. El índice [-1] accedería al último carácter de la cadena y el índice [-2] accedería al penúltimo carácter.

También puede acceder a una parte de una cadena, denominada segmento o subcadena. Esto le permite acceder a varios caracteres de una cadena. Puede hacer esto creando un rango, usando dos puntos como separador entre el inicio y el final del rango, como [2: 5]. Este rango es similar a la función range () que vimos anteriormente. Incluye el primer número, pero es uno menos que el último. También puede hacer referencia fácilmente al inicio o al final de la cadena dejando un valor en blanco. Por ejemplo, [: 5] nos daría todos los caracteres desde el principio hasta el cuarto carácter de la cadena. También podemos usar [5:] para obtener los caracteres desde el cuarto carácter hasta el final de la cadena.

Métodos básicos de cadena

En Python, las cadenas son inmutables. Esto significa que no se pueden modificar. Entonces, si quisiéramos corregir un error tipográfico en una cadena, no podemos simplemente modificar el carácter incorrecto. Tendríamos que crear una nueva cadena con el error tipográfico corregido. También podemos asignar un nuevo valor a la variable que contiene nuestra cadena.

Si no estamos seguros de cuál es el índice de nuestro error tipográfico, podemos usar el *índice del* método de cadena para localizarlo y devolver el índice. Imaginemos que tenemos la cadena **"leones, tigres y osos"** en la variable **animales** . Podemos localizar el índice que contiene la letra **g** usando *animals.index ("g")* , que devolverá el índice; en este caso 8. También podemos usar subcadenas para ubicar el índice donde comienza la subcadena. *animals.index ("osos")* devolvería 17, ya que ese es el comienzo de la subcadena. Si hay más de una coincidencia para una subcadena, el método de índice devolverá la primera coincidencia. Si intentamos localizar una subcadena que no existe en la cadena,explicando que no se encontró la subcadena.

Podemos evitar un ValueError comprobando primero si la subcadena existe en la cadena. Esto se puede hacer usando la palabra clave ***in*** . Vimos esta palabra clave antes cuando cubrimos bucles *for* . En este caso, es un condicional que será Verdadero o Falso. Si la subcadena se encuentra en la cadena, será Verdadero. Si la subcadena no se encuentra en la cadena, será False. Usando nuestra variable anterior **animales**, podemos hacer **"caballos" en animales** para comprobar si la subcadena "caballos" se encuentra en nuestra variable. En este caso, se evaluaría como Falso, ya que los caballos no están incluidos en nuestra cadena de ejemplo. Si hiciéramos **"tigres" en animales**, obtendríamos True, ya que esta subcadena está contenida en nuestra cadena.

Métodos de cadena avanzados

Ya hemos cubierto un montón de métodos de la clase String, así que sigamos construyéndolos y analicemos algunos métodos más avanzados.

El método de cadena **inferior** devolverá la cadena con todos los caracteres cambiados a minúsculas. Lo inverso de esto es el método **superior** , que devolverá la cadena en mayúsculas. Al igual que con los métodos anteriores, los llamamos en una cadena usando notación de puntos, como **"esto es una cadena" .upper ()** . Esto devolvería la cadena **"ESTO ES UNA CADENA"** . Esto puede ser muy útil al verificar la entrada del usuario, ya que alguien puede escribir todo en minúsculas, todo en mayúsculas o incluso una combinación de casos.

Puede utilizar el método de **tira** para eliminar los espacios en blanco circundantes de una cadena. Los espacios en blanco incluyen espacios, tabulaciones y caracteres de nueva línea. También puede usar los métodos  **lstrip** y **rstrip** para eliminar los espacios en blanco solo del lado izquierdo o derecho de la cadena, respectivamente.

El método **count** se puede usar para devolver el número de veces que una subcadena aparece en una cadena. Esto puede ser útil para saber cuántos caracteres aparecen en una cadena o contar el número de veces que aparece una determinada palabra en una oración o párrafo.

Si desea verificar si una cadena termina con una subcadena determinada, puede usar el método **endswith** . Esto devolverá True si la subcadena se encuentra al final de la cadena y False si no.

El método **isnumeric** puede verificar si una cadena está compuesta solo por números. Si la cadena contiene solo números, este método devolverá True. Podemos usar esto para verificar si una cadena contiene números antes de pasar la cadena a la función **int ()** para convertirla en un número entero, evitando un error. ¡Útil!

Echamos un vistazo a la concatenación de cadenas utilizando el signo más, anteriormente. También podemos usar el método de **unión** para concatenar cadenas. Este método se llama en una cadena que se utilizará para unir una lista de cadenas. El método toma una lista de cadenas para unir como parámetro y devuelve una nueva cadena compuesta de cada una de las cadenas de nuestra lista unidas usando la cadena inicial. Por ejemplo, **"" .join (["Esto", "es", "una", "oración"])** devolvería la cadena **"Esto es una oración"** .

La inversa del método de unión es el método de **división**. Esto nos permite dividir una cadena en una lista de cadenas. De forma predeterminada, se divide por cualquier carácter de espacio en blanco. También puede dividir por cualquier otro carácter pasando un parámetro.

Formato de cadena

Puede utilizar el método de **formato** en cadenas para concatenar y formatear cadenas de muchas formas eficaces. Para hacer esto, cree una cadena que contenga llaves, **{}** , como marcador de posición, para ser reemplazado. Luego llame al método de formato en la cadena usando *.format ()* y pase variables como parámetros. Las variables pasadas al método se utilizarán para reemplazar los marcadores de posición de los corchetes. Este método maneja automáticamente cualquier conversión entre tipos de datos por nosotros.

Si las llaves están vacías, se completarán con las variables pasadas en el orden en que se pasaron. Sin embargo, puede poner ciertas expresiones entre llaves para realizar operaciones de formateo de cadenas aún más poderosas. Puede poner el nombre de una variable entre corchetes y luego usar los nombres en los parámetros. Esto permite un código más fácil de leer y una mayor flexibilidad con el orden de las variables.

También puede poner una expresión de formato dentro de las llaves, lo que le permite modificar la forma en que se formatea la cadena. Por ejemplo, la expresión de formato **{: .2f}** significa que lo formatearías como un número flotante, con dos dígitos después del punto decimal. Los dos puntos actúan como un separador del nombre del campo, si ha especificado uno. También puede especificar la alineación del texto mediante el operador mayor que: **>** . Por ejemplo, la expresión **{:> 3.2f}** alinearía el texto tres espacios a la derecha, además de especificar un número flotante con dos lugares decimales. El formato de cadena puede ser muy útil para generar una salida textual fácil de leer.

# Hoja de referencia de cadenas

## **Hoja de referencia de cadenas**

En Python, hay muchas cosas que puede hacer con cadenas. En esta hoja de trucos, encontrará las operaciones de cadena y los métodos de cadena más comunes.

### **Operaciones de cadena**

* len (cadena) Devuelve la longitud de la cadena
* para el carácter de la cadena Itera sobre cada carácter de la cadena
* if substring in string Comprueba si la subcadena es parte de la cadena
* cadena [i] Accede al carácter en el índice i de la cadena, comenzando en cero
* cadena [i: j] Accede a la subcadena comenzando en el índice i, terminando en el índice j-1. Si se omite i, es 0 por defecto. Si se omite j, es len (cadena) por defecto.

### **Métodos de cadena**

* string.lower () / string.upper () Devuelve una copia de la cadena con todos los caracteres en minúsculas / mayúsculas
* string.lstrip () / string.rstrip () / string.strip () Devuelve una copia de la cadena sin espacios en blanco izquierda / derecha / izquierda o derecha
* string.count (substring) Devuelve el número de veces que la subcadena está presente en la cadena
* string.isnumeric () Devuelve True si solo hay caracteres numéricos en la cadena. Si no, devuelve False.
* string.isalpha () Devuelve True si solo hay caracteres alfabéticos en la cadena. Si no, devuelve False.
* string.split () / string.split (delimiter) Devuelve una lista de subcadenas que fueron separadas por espacios en blanco / delimitador
* string.replace (old, new) Devuelve una nueva cadena donde todas las apariciones de old han sido reemplazadas por nuevas.
* delimiter.join (lista de cadenas) Devuelve una nueva cadena con todas las cadenas unidas por el delimitador

Consulte la documentación oficial para conocer [todos los métodos String disponibles](https://docs.python.org/3/library/stdtypes.html#string-methods) .

# Hoja de referencia de formato de cadenas

## **Hoja de referencia de formato de cadenas**

Python ofrece diferentes formas de formatear cadenas. En el video, explicamos el método format (). En esta lectura, destacaremos tres formas diferentes de formatear cadenas. Para este curso solo necesitas conocer el método format (). Pero en Internet, puede encontrar cualquiera de los tres, por lo que es una buena idea saber que los demás existen.

### **Usando el método format ()**

El método de formato devuelve una copia de la cadena donde los marcadores de posición {} se han reemplazado con los valores de las variables. Estas variables se convierten en cadenas si aún no lo eran. Los marcadores de posición vacíos se reemplazan por las variables que se pasan al formato en el mismo orden.

# "base string with {} placeholders".format(variables)

example = "format() method"

formatted\_string = "this is an example of using the {} on a string".format(example)

print(formatted\_string)

"""Outputs:

this is an example of using the format() method on a string

"""

Si los marcadores de posición indican un número, se reemplazan por la variable correspondiente a ese orden (comenzando en cero).

 "{0} {1}".format(first, second)

first = "apple"

second = "banana"

third = "carrot"

formatted\_string = "{0} {2} {1}".format(first, second, third)

print(formatted\_string)

"""Outputs:

apple carrot banana

"""

Si los marcadores de posición indican un nombre de campo, se reemplazan por la variable correspondiente a ese nombre de campo. Esto significa que los parámetros a formatear deben pasarse indicando el nombre del campo.

# "{var1} {var2}".format(var1=value1, var2=value2)

"{:exp1} {:exp2}".format(value1, value2)

Si los marcadores de posición incluyen dos puntos, lo que viene después de los dos puntos es una expresión de formato. Consulte a continuación la referencia de la expresión.

Documentación oficial para [la sintaxis de cadena de formato](https://docs.python.org/3/library/string.html#formatstrings)

# {:d} integer value

'{:d}'.format(10.5) → '10'

### **Formatear expresiones**

| **Expr** | **Sentido** | **Ejemplo** |
| --- | --- | --- |
| {:re} | valor entero | '{: d}'. formato (10.5) → '10' |
| {: .2f} | punto flotante con tantos decimales | '{: .2f}'. Formato (0.5) → '0.50' |
| {: .2s} | cadena con tantos caracteres | '{: .2s}'. Formato ('Python') → 'Py' |
| {: <6 s} | cadena alineada a la izquierda tantos espacios | '{: <6s}'. Formato ('Py') → 'Py' |
| {:> 6 s} | cadena alineada a la derecha tantos espacios | '{:> 6s}'. Formato ('Py') → 'Py' |
| {: ^ 6s} | cadena centrada en tantos espacios | '{: ^ 6s}'. Formato ('Py') → 'Py' |

Consulte la documentación oficial para conocer [todas las expresiones disponibles](https://docs.python.org/3/library/string.html#format-specification-mini-language) .

### **Formato de cadena antiguo (opcional)**

El método format () se introdujo en Python 2.6. Antes de eso, el operador% (módulo) podría usarse para obtener un resultado similar. Si bien este método **ya no se recomienda** para código nuevo, es posible que lo encuentre en el código de otra persona. Esto es lo que parece:

"cadena base con% s marcador de posición"% variable

El operador% (módulo) devuelve una copia de la cadena donde los marcadores de posición indicados por% seguidos de una expresión de formato se reemplazan por las variables después del operador.

"cadena base con% dy% d marcadores de posición"% (valor1, valor2)

Para reemplazar más de un valor, los valores deben escribirse entre paréntesis. La expresión de formato debe coincidir con el tipo de valor.

"% (var1)% (var2)"% {var1: value1, var2: value2}

Las variables se pueden reemplazar por el nombre usando una sintaxis de diccionario (aprenderemos sobre los diccionarios en un próximo video).

"Artículo:% s - Monto:% d - Precio:% ​​.2f"% (artículo, monto, precio)

Las expresiones de formato son en su mayoría las mismas que las del método format ().

Consulte la documentación oficial para conocer el [formato de cadena antiguo](https://docs.python.org/3/library/stdtypes.html#old-string-formatting) .

### **Literales de cadena formateados (opcional)**

Esta función se agregó en Python 3.6 y aún no se usa mucho. Nuevamente, está incluido aquí en caso de que lo encuentre en el futuro, pero no es necesario para este ni ningún otro curso futuro.

Un literal de cadena formateado o una cadena f es una cadena que comienza con 'f' o 'F' antes de las comillas. Estas cadenas pueden contener {} marcadores de posición que utilizan expresiones como las que se utilizan para las cadenas de métodos de formato.

La diferencia importante con el método de formato es que toma el valor de las variables del contexto actual, en lugar de tomar los valores de los parámetros.

Ejemplos:

>>> nombre = "Micah"

>>> print (f'Hola {nombre} ')

Hola micah

>>> item = "Taza morada"

>>> cantidad = 5

>>> precio = monto \* 3.25

>>> print (f'Item: {item} - Importe: {amount} - Precio: {price: .2f} ')

Artículo: Copa Morada - Cantidad: 5 - Precio: 16.25

Consulte la documentación oficial para [f-strings](https://docs.python.org/3/reference/lexical_analysis.html#f-strings) .

# Listas definidas

Las listas en Python se definen usando corchetes, con los elementos almacenados en la lista separados por comas: **list = ["This", "is", "a", "list"]** . Puede usar la función **len ()** para devolver el número de elementos en una lista: **len (lista)** devolvería **4** . También puede utilizar la palabra clave **in** para comprobar si una lista contiene un elemento determinado. Si el elemento está presente, devolverá un booleano verdadero. Si el elemento no se encuentra en la lista, devolverá False. Por ejemplo, **"Esto" en la lista** devolvería True en nuestro ejemplo. De manera similar a las cadenas, las listas también pueden usar la indexación para acceder a elementos específicos en una lista según su posición., que le permitiría acceder a la cadena **"Esto"** .

En Python, las listas y las cadenas son bastante similares. Ambos son ejemplos de secuencias de datos. Las secuencias tienen propiedades similares, como (1) poder iterar sobre ellas usando **bucles for** ; (2) indexación de soporte; (3) usar la función **len** para encontrar la longitud de la secuencia; (4) usar el operador más **+** para concatenar; y (5) usar la palabra clave **in** para verificar si la secuencia contiene un valor. Comprender estos conceptos le permite aplicarlos también a otros tipos de secuencia.

# Modificar listas

Si bien las listas y las cadenas son secuencias, una gran diferencia entre ellas es que las listas son mutables. Esto significa que el contenido de la lista se puede cambiar, a diferencia de las cadenas, que son inmutables. Puede agregar, eliminar o modificar elementos en una lista.

Puede agregar elementos al final de una lista utilizando el método de **adición** . Llamas a este método en una lista usando notación de puntos y pasas el elemento que se agregará como parámetro. Por ejemplo, **list.append ("Datos nuevos")** agregaría la cadena "Datos nuevos" al final de la lista llamada lista.

Si desea agregar un elemento a una lista en una posición específica, puede usar el método **insert** . El método toma dos parámetros: el primero especifica el índice en la lista y el segundo es el elemento que se agregará a la lista. Entonces **list.insert (0, "Datos nuevos")** agregaría la cadena "Datos nuevos" al **principio** de la lista. Esto no sobrescribirá el elemento existente al principio de la lista. Simplemente cambiaría todos los demás elementos en uno. Si especifica un índice que es mayor que la longitud de la lista, el elemento simplemente se agregará al final de la lista.

Puede eliminar elementos de la lista mediante el método de **eliminación** . Este método toma un elemento como parámetro y elimina la primera aparición del elemento. Si el elemento no se encuentra en la lista, obtendrá un error **ValueError** que explica que el elemento no se encontró en la lista.

También puede eliminar elementos de una lista mediante el método **pop** . Este método difiere del método remove en que toma un índice como parámetro y devuelve el elemento que se eliminó. Esto puede resultar útil si no sabe cuál es el valor, pero sabe dónde se encuentra. Esto también puede ser útil cuando necesita acceder a los datos y también desea eliminarlos de la lista.

Finalmente, puede cambiar un elemento en una lista usando la indexación para sobrescribir el valor almacenado en el índice especificado. Por ejemplo, puede ingresar **list [0] = "Datos antiguos"** para sobrescribir el primer elemento de una lista con la nueva cadena "Datos antiguos".

# Tuplas

Como mencionamos anteriormente, las cadenas y las listas son ejemplos de secuencias. Las cadenas son secuencias de caracteres y son inmutables. Las listas son secuencias de elementos de cualquier tipo de datos y son mutables. El tercer tipo de secuencia es la tupla. Las tuplas son como listas, ya que pueden contener elementos de cualquier tipo de datos. Pero a diferencia de las listas, las tuplas son inmutables. Se especifican utilizando paréntesis en lugar de corchetes.

Quizás se pregunte por qué las tuplas son una cosa, dado lo similares que son a las listas. Las tuplas pueden ser útiles cuando necesitamos asegurarnos de que un elemento está en una posición determinada y no cambiará. Dado que las listas son mutables, podemos cambiar el orden de los elementos. Dado que el orden de los elementos en una tupla no se puede cambiar, la posición del elemento en una tupla puede tener significado. Un buen ejemplo de esto es cuando una función devuelve varios valores. En este caso, lo que se devuelve es una tupla, con los valores de retorno como elementos de la tupla. El orden de los valores devueltos es importante y una tupla asegura que el orden no va a cambiar. El almacenamiento de los elementos de una tupla en variables independientes se denomina desempaquetado. Esto le permite tomar múltiples valores devueltos de una función y almacenar cada valor en su propia variable.

# Iterar sobre listas usando Enumerate

Cuando cubrimos los bucles for , mostramos el ejemplo de iterar sobre una lista. Esto le permite iterar sobre cada elemento de la lista, exponiendo el elemento al bucle for como una variable. Pero, ¿qué sucede si desea acceder a los elementos de una lista, junto con el índice del elemento en cuestión? Puede hacer esto usando la función **enumerate ()** . La función enumerate () toma una lista como parámetro y devuelve una tupla para cada elemento de la lista. El primer valor de la tupla es el índice y el segundo valor es el elemento en sí.

# Lista de comprensiones

Puede crear listas a partir de secuencias usando un bucle for, pero hay una forma más simplificada de hacerlo: **comprensión de listas** . Las listas por comprensión le permiten crear una nueva lista a partir de una secuencia o un rango en una sola línea.

Por ejemplo, **[x \* 2 para x en el rango (1,11)]** es una simple lista de comprensión. Esto iteraría sobre el rango de 1 a 10 y multiplicaría cada elemento en el rango por 2. Esto daría como resultado una lista de múltiplos de 2, de 2 a 20.

También puede usar condicionales con listas por comprensión para construir declaraciones aún más complejas y poderosas. Puede hacer esto agregando una declaración if al final de la comprensión. Por ejemplo, **[x para x en el rango (1,101) si x% 10 == 0]** generaría una lista que contiene todos los números enteros divisibles por 10 de 1 a 100. La declaración if que agregamos aquí evalúa cada valor en el rango de 1 a 100 para comprobar si es divisible por 10. Si lo es, se agrega a la lista.

Las listas por comprensión pueden ser muy poderosas, pero también pueden ser muy complejas, lo que da como resultado un código difícil de leer. Tenga cuidado al usarlos, ya que podría dificultar que otra persona que mire su código entienda fácilmente lo que está haciendo.

# Hoja de referencia de operaciones de listas y tuplas

## **Hoja de referencia de operaciones de listas y tuplas**

Tanto las listas como las tuplas son secuencias, por lo que comparten una serie de operaciones de secuencia. Pero, debido a que las listas son mutables, también hay una serie de métodos específicos solo para las listas. Esta hoja de trucos le ofrece un resumen de las operaciones comunes primero y, en segundo lugar, de las operaciones específicas de la lista.

### **Operaciones de secuencia común**

* len (secuencia) Devuelve la longitud de la secuencia
* para elemento en secuencia Itera sobre cada elemento en la secuencia
* si el elemento en secuencia Comprueba si el elemento es parte de la secuencia
* secuencia [i] Accede al elemento en el índice i de la secuencia, comenzando en cero
* secuencia [i: j] Accede a un segmento que comienza en el índice i y termina en el índice j-1. Si se omite i, es 0 por defecto. Si se omite j, es len (secuencia) por defecto.
* para índice, elemento en enumerar (secuencia) Repite tanto los índices como los elementos de la secuencia al mismo tiempo

Consulte la [documentación oficial para las operaciones de secuencia](https://docs.python.org/3/library/stdtypes.html#sequence-types-list-tuple-range) .

### **Listar operaciones y métodos específicos**

* lista [i] = x Reemplaza el elemento en el índice i con x
* list.append (x) Inserta x al final de la lista
* list.insert (i, x) Inserta x en el índice i
* list.pop (i) Devuelve el elemento un índice i, también eliminándolo de la lista. Si se omite i, el último elemento se devuelve y se elimina.
* list.remove (x) Elimina la primera aparición de x en la lista
* list.sort () Ordena los elementos de la lista
* list.reverse () Invierte el orden de los elementos de la lista
* list.clear () Elimina todos los elementos de la lista
* list.copy () Crea una copia de la lista
* list.extend (other\_list) Agrega todos los elementos de other\_list al final de la lista

La mayoría de estos métodos provienen del hecho de que las listas son secuencias mutables. Para obtener más información, consulte la [documentación oficial para secuencias mutables](https://docs.python.org/3/library/stdtypes.html#mutable-sequence-types) y la [documentación específica de](https://docs.python.org/3/library/stdtypes.html#lists) la [lista](https://docs.python.org/3/library/stdtypes.html#lists) .

### **Comprensión de listas**

* [expresión para variable en secuencia] Crea una nueva lista basada en la secuencia dada. Cada elemento es el resultado de la expresión dada.
* [expresión para variable en secuencia si condición] Crea una nueva lista basada en la secuencia dada. Cada elemento es el resultado de la expresión dada; los elementos solo se agregan si la condición es verdadera.

# Definición de diccionarios

Los diccionarios son otra estructura de datos en Python. Son similares a una lista en el sentido de que se pueden usar para organizar datos en colecciones. Sin embargo, no se accede a los datos de un diccionario en función de su posición. Los datos de un diccionario se organizan en pares de claves y valores. Utiliza la clave para acceder al valor correspondiente. Cuando un índice de lista es siempre un número, una clave de diccionario puede ser de un tipo de datos diferente, como una cadena, un entero, un flotante o incluso una tupla.

Al crear un diccionario, usa llaves: **{}** . Al almacenar valores en un diccionario, primero se especifica la clave, seguida del valor correspondiente, separado por dos puntos. Por ejemplo, **animales = {"osos": 10, "leones": 1, "tigres": 2}** crea un diccionario con tres pares de valores clave, almacenados en la variable animales. La clave "osos" apunta al valor entero 10, mientras que la clave "leones" apunta al valor entero 1 y "tigres" apunta al entero 2. Puedes acceder a los valores haciendo referencia a la clave, así: **animales [ "osos"]**. Esto devolvería el entero 10, ya que ese es el valor correspondiente para esta clave.

También puede verificar si una clave está contenida en un diccionario usando la palabra clave **in**. Al igual que otros usos de esta palabra clave, devolverá True si la clave se encuentra en el diccionario; de lo contrario, devolverá False.

Los diccionarios son mutables, lo que significa que se pueden modificar agregando, quitando y reemplazando elementos en un diccionario, similar a las listas. Puede agregar un nuevo par clave-valor a un diccionario asignando un valor a la clave, como este: **animales ["cebras"] = 2**. Esto crea la nueva clave en el diccionario de animales llamada cebras y almacena el valor 2. Puede modificar el valor de una clave existente haciendo lo mismo. Entonces, **animals ["bears"] = 11** cambiaría el valor almacenado en la clave bears de 10 a 11. Por último, puede eliminar elementos de un diccionario usando la palabra clave **del**. Al hacer **del animals ["leones"]**, eliminaría el par clave-valor del diccionario de animales.

# Iterando sobre diccionarios

Puede iterar sobre diccionarios usando un bucle for , al igual que con cadenas, listas y tuplas. Esto iterará sobre la secuencia de claves en el diccionario. Si desea acceder a los valores correspondientes asociados con las claves, puede utilizar las claves como índices. O puede usar el método de **elementos** en el diccionario, como **dictionary.items ()** . Este método devuelve una tupla para cada elemento del diccionario, donde el primer elemento de la tupla es la clave y el segundo es el valor.

Si solo desea acceder a las claves de un diccionario, puede utilizar el método **keys ()** en el diccionario: **dictionary.keys ()** . Si solo desea los valores, puede usar el método **values ​​()** : **dictionary.values ​​()** .

# Hoja de referencia de métodos de diccionario

## **Hoja de referencia de métodos de diccionario**

**Definición**

x = {clave1: valor1, clave2: valor2}

**Operaciones**

* len (diccionario): devuelve el número de elementos del diccionario
* for key in dictionary: itera sobre cada clave del diccionario
* para clave, valor en dictionary.items (): repite cada par de clave y valor en el diccionario
* if key in dictionary: comprueba si la clave está en el diccionario
* diccionario [clave]: accede al elemento con la clave clave del diccionario
* diccionario [clave] = valor: establece el valor asociado con la clave
* del dictionary [clave]: elimina el elemento con la clave clave del diccionario

**Métodos**

* dict.get (clave, predeterminado): devuelve el elemento correspondiente a la clave, o predeterminado si no está presente
* dict.keys (): devuelve una secuencia que contiene las claves en el diccionario
* dict.values ​​(): devuelve una secuencia que contiene los valores en el diccionario
* dict.update (other\_dictionary): actualiza el diccionario con los elementos que provienen del otro diccionario. Se reemplazarán las entradas existentes; se agregarán nuevas entradas.
* dict.clear (): elimina todos los elementos del diccionario

Consulte la [documentación oficial para conocer las operaciones y los métodos del diccionario](https://docs.python.org/3/library/stdtypes.html#mapping-types-dict) .

# Definición de programación orientada a objetos

En la programación orientada a objetos, los conceptos se modelan como clases y objetos. Una idea se define mediante una clase y una instancia de esta clase se denomina objeto. Casi todo en Python es un objeto, incluidas cadenas, listas, diccionarios y números. Cuando creamos una lista en Python, estamos creando un objeto que es una instancia de la clase list, que representa el concepto de una lista. Las clases también tienen atributos y métodos asociados. Los atributos son las características de la clase, mientras que los métodos son funciones que forman parte de la clase.

# Clases y objetos en detalle

Podemos usar la función **type ()** para averiguar a qué clase pertenece una variable o valor. Por ejemplo, el **tipo ("")** nos dice que se trata de una clase de cadena. El único atributo en este caso es el valor de la cadena, pero hay un montón de métodos asociados con la clase. Hemos visto el método **upper ()** , que devuelve la cadena en mayúsculas, así como **isnumeric ()** que devuelve un booleano que nos dice si la cadena es un número o no. Puedes usar **dir ()**función para imprimir todos los atributos y métodos de un objeto. Cada cadena es una instancia de la clase de cadena y tiene los mismos métodos de la clase principal. Dado que el contenido de la cadena es diferente, los métodos devolverán valores diferentes. También puede utilizar la función **help ()** en un objeto, que devolverá la documentación de la clase correspondiente. Esto mostrará todos los métodos de la clase, junto con los parámetros que reciben los métodos, los tipos de valores devueltos y una descripción de los métodos.

Definición de clases (opcional)

Podemos crear y definir nuestras clases en Python de forma similar a como definimos funciones. Comenzamos con la palabra clave **class** , seguida del nombre de nuestra clase y dos puntos. Las pautas de estilo de Python recomiendan que los nombres de las clases comiencen con una letra mayúscula. Después de la línea de definición de clase está el cuerpo de la clase, sangrado a la derecha. Dentro del cuerpo de la clase, podemos definir atributos para la clase.

Tomemos nuestro ejemplo de clase de Apple:

>>> class Apple:

...     color = ""

...     flavor = ""

...

Podemos crear una nueva instancia de nuestra nueva clase asignándola a una variable. Esto se hace llamando al nombre de la clase como si fuera una función. Podemos establecer los atributos de nuestra instancia de clase accediendo a ellos usando la notación de puntos. La notación de puntos se puede utilizar para establecer o recuperar atributos de objetos, así como también llamar a métodos asociados con la clase.

>>> jonagold = Apple()

>>> jonagold.color = "red"

>>> jonagold.flavor = "sweet"

Creamos una instancia de Apple llamada jonagold y configuramos los atributos de color y sabor para este objeto de Apple. Podemos crear otra instancia de una manzana y establecer diferentes atributos para diferenciar entre dos variedades diferentes de manzanas.

>>> golden = Apple()

>>> golden.color = "Yellow"

>>> golden.flavor = "Soft"

Ahora tenemos otro objeto de Apple llamado dorado que también tiene atributos de color y sabor. Pero estos atributos tienen valores diferentes.

¿Qué es un método?

Llamar a métodos sobre objetos ejecuta funciones que operan sobre atributos de una instancia específica de la clase. Esto significa que llamar a un método en una lista, por ejemplo, solo modifica esa instancia de una lista, y no todas las listas globalmente. Podemos definir métodos dentro de una clase creando funciones dentro de la definición de la clase. Estos métodos de instancia pueden tomar un parámetro llamado **self,** que representa la instancia en la que se está ejecutando el método. Esto le permitirá acceder a los atributos de la instancia usando notación de puntos, como **self.name** , que accederá al atributo de nombre de esa instancia específica del objeto de clase. Cuando tiene variables que contienen diferentes valores para diferentes instancias, estas se denominan variables de instancia.

Métodos especiales

En lugar de crear clases con valores vacíos o predeterminados, podemos establecer estos valores cuando creamos la instancia. Esto asegura que no perdamos un valor importante y evita muchas líneas de código innecesarias. Para hacer esto, usamos un método especial llamado **constructor** . A continuación se muestra un ejemplo de una clase de Apple con un método de constructor definido.

>>> class Apple:

...     def \_\_init\_\_(self, color, flavor):

...         self.color = color

...         self.flavor = flavor

Cuando llamas el nombre de una clase, se llama al constructor de esa clase. Este método constructor siempre se llama **\_\_init\_\_** . Quizás recuerde que los métodos especiales comienzan y terminan con dos caracteres de subrayado. En nuestro ejemplo anterior, el método constructor toma la variable self, que representa la instancia, así como los parámetros de color y sabor. A continuación, el método constructor utiliza estos parámetros para establecer los valores de la instancia actual. Entonces, ahora podemos crear una nueva instancia de la clase Apple y establecer los valores de color y sabor, todo listo:

>>> jonagold = Apple("red", "sweet")

>>> print(jonagold.color)

Red

Además del método especial del constructor **\_\_init\_\_** , también existe el método especial **\_\_str\_\_** . Este método nos permite definir cómo se imprimirá una instancia de un objeto cuando se pase a la función print (). Si un objeto no tiene este método especial definido, terminará usando la representación predeterminada, que imprimirá la posición del objeto en la memoria. No es muy útil. Aquí está nuestra clase de Apple, con el método **\_\_str\_\_** agregado:

>>> class Apple:

...     def \_\_init\_\_(self, color, flavor):

...         self.color = color

...         self.flavor = flavor

...     def \_\_str\_\_(self):

...         return "This apple is {} and its flavor is {}".format(self.color, self.flavor)

...

Ahora, cuando pasamos un objeto de Apple a la función de impresión, obtenemos una bonita cadena formateada:

>>> jonagold = Apple("red", "sweet")

>>> print(jonagold)

This apple is red and its flavor is sweet

Esta manzana es roja y su sabor es dulce

Es una buena práctica pensar en cómo se podría usar tu clase y definir un método \_\_str\_\_ al crear objetos que quizás quieras imprimir más tarde.

# Herencia de objetos

En la programación orientada a objetos, el concepto de herencia le permite construir relaciones entre objetos, agrupando conceptos similares y reduciendo la duplicación de código. Creemos una clase de fruta personalizada con atributos de color y sabor:

>>> class Fruit:

...     def \_\_init\_\_(self, color, flavor):

...         self.color = color

...         self.flavor = flavor

...

Definimos una clase Fruit con un constructor para atributos de color y sabor. A continuación, definiremos una clase Apple junto con una nueva clase Grape, las cuales queremos heredar propiedades y comportamientos de la clase Fruit:

>>> class Apple(Fruit):

...     pass

...

>>> class Grape(Fruit):

...     pass

...

En Python, usamos paréntesis en la declaración de la clase para que la clase herede de la clase Fruit. Entonces, en este ejemplo, le estamos indicando a nuestra computadora que tanto la clase Apple como la clase Grape heredan de la clase Fruit. Esto significa que ambos tienen el mismo método constructor que establece los atributos de color y sabor. Ahora podemos crear instancias de nuestras clases Apple y Grape:

>>> granny\_smith = Apple("green", "tart")

>>> carnelian = Grape("purple", "sweet")

>>> print(granny\_smith.flavor)

tart

>>> print(carnelian.color)

purple

La herencia nos permite definir atributos o métodos que son compartidos por todos los tipos de fruta sin tener que definirlos en cada clase de fruta de forma individual. Luego, también podemos definir atributos o métodos específicos que solo son relevantes para un tipo específico de fruta. Veamos otro ejemplo, esta vez con animales:

>>> class Animal:

...     sound = ""

...     def \_\_init\_\_(self, name):

...         self.name = name

...     def speak(self):

...         print("{sound} I'm {name}! {sound}".format(

...             name=self.name, sound=self.sound))

...

>>> class Piglet(Animal):

...     sound = "Oink!"

...

>>> class Cow(Animal):

...     sound = "Moooo"

...

Definimos una clase padre, Animal, con dos tipos de animales que heredan de esa clase: Lechón y Vaca. La clase Animal padre tiene un atributo para almacenar el sonido que hace el animal y la clase constructora toma el nombre que se asignará a la instancia cuando se cree. También existe el método de hablar, que imprime el nombre del animal junto con el sonido que hace. Definimos las clases Piglet y Cow, que heredan de la clase Animal, y configuramos los atributos de sonido para cada tipo de animal. Ahora, podemos crear instancias de nuestras clases Piglet y Cow y hacer que hablen:

>>> hamlet = Piglet("Hamlet")

>>> hamlet.speak()

Oink! I'm Hamlet! Oink!

...

>>> class Cow(Animal):

...     sound = "Moooo"

...

>>> milky = Cow("Milky White")

>>> milky.speak()

Moooo I'm Milky White! Moooo

Creamos instancias de la clase Piglet y Cow, y establecemos los nombres para nuestras instancias. Luego llamamos al método speak de cada instancia, lo que da como resultado la impresión de la cadena formateada; incluye el sonido que hace el tipo de animal, junto con el nombre de instancia que le asignamos.

# Composición del objeto

Puede tener una situación en la que dos clases diferentes estén relacionadas, pero no haya herencia. Esto se conoce como **composición** , donde una clase hace uso de código contenido en otra clase. Por ejemplo, imagine que tenemos una clase **Package** que representa un paquete de software. Contiene atributos sobre el paquete de software, como el nombre, la versión y el tamaño. También tenemos una clase **Repository** que representa todos los paquetes disponibles para su instalación. Si bien no existe una relación de herencia entre las dos clases, están relacionadas. La clase Repository contendrá un diccionario o lista de paquetes que están contenidos en el repositorio. Echemos un vistazo a una definición de clase de repositorio de ejemplo:

>>> class Repository:

...      def \_\_init\_\_(self):

...          self.packages = {}

...      def add\_package(self, package):

...          self.packages[package.name] = package

...      def total\_size(self):

...          result = 0

...          for package in self.packages.values():

...              result += package.size

...          return result

En el método constructor, inicializamos el diccionario de paquetes, que contendrá los objetos del paquete disponibles en esta instancia del repositorio. Inicializamos el diccionario en el constructor para asegurarnos de que cada instancia de la clase Repository tenga su propio diccionario.

Luego definimos el método add\_package, que toma un objeto Package como parámetro, y luego lo agrega a nuestro diccionario, usando el atributo del nombre del paquete como clave.

Finalmente, definimos un método total\_size que calcula el tamaño total de todos los paquetes contenidos en nuestro repositorio. Este método itera a través de los valores en nuestro diccionario de repositorio y suma los atributos de tamaño de cada objeto de paquete contenido en el diccionario, devolviendo el total al final. En este ejemplo, estamos haciendo uso de los atributos del paquete dentro de nuestra clase Repository. También estamos llamando al método values ​​() en nuestra instancia de diccionario de paquetes. La composición nos permite utilizar objetos como atributos, así como acceder a todos sus atributos y métodos.

# Aumento de Python con módulos

Los módulos de Python son archivos separados que contienen clases, funciones y otros datos que nos permiten importar y hacer uso de estos métodos y clases en nuestro propio código. Python viene con muchos módulos listos para usar. Estos módulos se conocen como la biblioteca estándar de Python. Puede hacer uso de estos módulos utilizando la palabra clave **import** , seguida del nombre del módulo. Por ejemplo, importaremos el módulo **aleatorio** y luego llamaremos a la función **randint** dentro de este módulo:

>>> import random

>>> random.randint(1,10)

8

>>> random.randint(1,10)

7

>>> random.randint(1,10)

1

sta función toma dos parámetros enteros y devuelve un entero aleatorio entre los valores que le pasamos; en este caso, 1 y 10. Puede notar que llamar a funciones en un módulo es muy similar a llamar a métodos en una clase. Aquí también usamos la notación de puntos, con un punto entre el módulo y los nombres de las funciones.

Echemos un vistazo a otro módulo: **datetime** . Este módulo es muy útil cuando se trabaja con fechas y horas.

>>> import datetime

>>> now = datetime.datetime.now()

>>> type(now)

<class 'datetime.datetime'>

>>> print(now)

2019-04-24 16:54:55.155199

Primero, importamos el módulo. A continuación, llamamos al método **now ()** que pertenece a la clase **datetime** contenida en el módulo **datetime** . Este método genera una instancia de la clase datetime para la fecha y hora actuales. Esta instancia tiene algunos métodos que podemos llamar:

>>> print(now)

2019-04-24 16:54:55.155199

>>> now.year

2019

>>> print(now + datetime.timedelta(days=28))

2019-05-22 16:54:55.155199

Cuando llamamos a la función de impresión con una instancia de la clase datetime, obtenemos la fecha y la hora impresas en un formato específico. Esto se debe a que la clase datetime tiene definido un método **\_\_str\_\_** que genera la cadena formateada que vemos aquí. También podemos llamar directamente a los atributos y métodos de la clase, como con **now.year** que devuelve el atributo de año de la instancia.

Por último, podemos acceder a otras clases contenidas en el módulo datetime, como la clase **timedelta** . En este ejemplo, estamos creando una instancia de la clase timedelta con el parámetro de 28 días. Luego agregamos este objeto a nuestra instancia de la clase datetime de antes e imprimimos el resultado. Esto tiene el efecto de agregar 28 días a nuestro objeto de fecha y hora original.