Resumen Python 3

Víctor Mardones Bravo

Febrero de 2021

Índice general

Capítulo 1

Introducción a Python

1.1. ¿Qué es Python?

Python es un lenguaje de programación de alto nivel, con aplicaciones en numerosas áreas, incluyendo programación web, scripting, computación científica e inteligencia artificial.

Es muy popular y usado por organizaciones como Google, la NASA, la CIA y Disney.

No hay limitaciones en lo que se puede construir usando Python. Esto incluye aplicaciones autónomas, aplicaciones web, juegos, ciencia de datos, modelos de machine learning y mucho más.

Dato curioso: Según el creador Guido van Rossum, el nombre de Python viene de la serie de comedia británica "El Circo Volador de Monty Python".

1.2. Hola mundo

Para mostrar el texto "Hola mundo" en pantalla se puede usar la función print().

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

Salida:

```
Hola mundo
```

Cada declaración de impresión print() genera texto en una nueva línea.

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

Salida:

```
Hola
Mundo
```

1.3. El Zen de Python

El Zen de Python es una colección de 19 "principios" para escribir programas de computadores que influenciaron el diseño y representan la filosofía de este lenguaje de programación.

Se mostrará en pantalla la primera vez que se ejecute la siguiente linea, para importar la librería "this".

```
1 import this
```

Después se mostrará el siguiente texto.

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Capítulo 2

Conceptos básicos

2.1. Comentarios

Los comentarios son anotaciones en el código utilizadas para hacerlo más fácil de entender. No afectan la ejecución del código.

En Python, los comentarios comienzan con el símbolo #. Todo el texto luego de este # (dentro de la misma línea) es ignorado.

```
# Este es un comentario
```

Python no soporta comentarios multilínea, al contrario de otros lenguajes de programación.

A lo largo de este resumen, se usarán comentarios para mostrar la salida de algunas funciones, cuando sea posible.

2.2. Operaciones aritméticas

Python tiene la capacidad de realizar cálculos. Los operadores +, -, * y / representan suma, resta, multiplicación y división, respectivamente.

```
print(1 + 1)  # suma
print(5 - 2)  # resta
print(2 * 2)  # multiplicación
print(4 / 4)  # división
```

Salida:

```
2
3
4
1.0
```

Los espacios entre los signos y los números son opcionales, pero hacen que el código sea más fácil de leer.

2.3. La regla PEMDAS

Las operaciones en Python siguen el orden dado por la regla PEMDAS:

1. Paréntesis
2. Exponentes
3. Multiplicación y división (de izquierda a derecha)
4. Adición y Sustracción (de izquierda a derecha)

```
print(2*7 - 1 + 4/2*3)
# 2*7 - 1 + 4/2*3
# 14 - 1 + 6.0
# 19.0
```

Salida:

```
19.0
```

2.4. Paréntesis

Se pueden usar paréntesis () para agrupar operaciones y hacer que estas se realicen primero, siguiendo la regla PEMDAS.

```
print((2 + 3) * 4)
# (2 + 3) * 4
# 5 * 4
# 20
```

Salida:

```
20
```

2.5. Floats

Para representar números racionales o que no son enteros, se usa el tipo de dato float o punto flotante. Se pueden crear directamente ingresando un número con un punto decimal, o como resultado de una división.

```
print(0.25)
print(3 / 4)
```

Salida:

```
0.25
0.75
```

Se debe tener en cuenta que los computadores no pueden almacenar perfectamente el valor de los floats, lo cual a menudo conduce a errores.

```
print(0.1 + 0.2)
```

Salida:

```
print(0.1 + 0.2)
```

El error mostrado arriba es un error clásico de la aritmética de punto flotante. Incluso tiene su propio sitio web.

Al trabajar con floats, no es necesario escribir un 0 a la izquierda del punto decimal.

```
print(.42)
```

Salida:

```
0.42
```

Esta notación se asemeja a decir "punto cinco" en vez de "cero punto cinco".

El resultado de cualquier operación entre floats o entre un float y un entero siempre dará como resultado un float. La división entre enteros también da como resultado un float.

```
print(1 + 2.0)
print(15.0 - 7)
print(3 * 4.0)
print(10 / 2)
print(10.0 / 2)
```

Salida:

```
3.0
8.0
12.0
5.0
5.0
```

2.6. Exponenciación

Otra operación soportada es la exponenciación, que es la elevación de un número a la potencia de otro. Esto se realiza usando el operador **.

Ejemplo equivalente a $2^5 = 32$.

```
print(2**5)
```

Salida:

```
32
```

2.7. Cociente y resto

La división entera se realiza usando el operador //, donde el resultado es la parte entera que queda al realizar la división, también conocida como cociente.

La división entera retorna un entero en vez de un float.

```
print(7 // 2)
```

Salida:

```
3
```

Para obtener el resto al realizar una división entera, se debe usar el operador módulo %.

```
print(7 % 2)
```

Salida:

```
1
```

Esta operación es equivalente a $7 \mod 2$ en aritmética modular.

Este operador viene de la aritmética modular, y uno de sus usos más comunes es para saber si un número es múltiplo de otro. Esto se hace revisando si el módulo al dividirlo por ese otro número es 0.

```
print(10 % 2)
print(15 % 3)
print(16 % 7)
```

Salida:

```
0
0
2
```

En el caso mostrado anteriormente, se infiere que 10 es múltiplo de 2, que 15 es múltiplo de 3 y que 16 no es múltiplo de 7. El caso particular módulo 2 también puede usarse para saber si un número es par o no.

Capítulo 3

Cadenas de texto

3.1. Strings o cadenas de caracteres

Las cadenas de caracteres se crean introduciendo el texto entre comillas simples ' ' o dobles " ".

```
print('texto')
print("texto")
```

Salida:

```
texto
texto
```

Un string debe empezar y terminar con comillas del mismo tipo, no se permiten comillas mixtas.

```
# String no válido
print('texto")
```

Salida:

```
SyntaxError: EOL while scanning string literal
```

```
# String no válido
print("texto')
```

Salida:

```
SyntaxError: EOL while scanning string literal
```

3.2. Cadena vacía

A veces es necesario inicializa un string, pero sin agregarle información. Una cadena vacía es definida como '' o "".

```
cadena_vacia1 = ''
cadena_vacia2 = ""
```

Estos strings vacíos se inicializan en variables, lo cual se verá en el capítulo siguiente.

3.3. Caracteres especiales

Algunos caracteres no se pueden incluir directamente en una cadena. Para esos casos, se debe incluir la barra diagonal inversa \antes de ellos.

```
print('\'')  # '
print('\"')  # "
print('\\')  # \
```

Salida:

```
'
"
\
```

Los caracteres ′, ″ y \son especiales, porque normalmente cumplen funciones especiales dentro de strings.

Si el string se define entre comillas dobles, no es necesario poner ' para ingresar comillas simples dentro de él, y viceversa.

```
print("This isn't spanish")
print('Hola "mundo"')
```

Salida:

```
This isn't spanish
Hola "mundo"
```

3.4. Secuencias de escape

Las secuencias de escape también se pueden incluir usando el símbolo \dentro de cadenas de texto. Su origen viene de las secuencias de escape usadas en las máquinas de escribir.

Algunas de las secuencias de escape más usadas son:

- Nueva línea (new line): Avanza una línea hacia adelante (salto de línea) y deja el cursor al principio de esta línea (retorno de carro). Representado por \n.

```
print('Hola\nMundo')
```

Salida:

```
Hola
Mundo
```

Cualquier caracter después de \n queda en la línea siguiente.

- Tabulador horizontal (horizontal tab): Añade un salto de tabulador horizontal. Representado por \t.

```
print('1\t2\t3\t4')
```

Salida:

```
1       2       3       4
```

El salto de tabulador avanza hasta el siguiente "tab stop" de la misma línea.

- Retorno de carro (carriage return): Mueve el "carro" (cursor) al principio de la línea actual. Esto permite sobreescribir los caracteres escritos anteriormente. Representado por \r.

```
print('12345\r67')
```

Salida:

```
67345
```

En algunos intérpretes, borra la línea además de volver al inicio.

- Retroceso (backspace): Borra el último carácter y mueve el cursor al carácter anterior. Representado por \b.

```
print('123\b45')
```

Salida:

```
1245
```

Otras secuencias de escape que cada vez se usan menos son:

- Tabulador vertical (vertical tab): Añade un salto de tabulador vertical. Representado por \v.

```
print('Hola\vmundo')
```

La tabulación vertical avanza hasta la siguiente línea que sea una "tab stop".

- Salto de página (form feed): Baja a la próxima "página". Representado por \f.

```
print('Hola\fmundo')
```

Algunos programadores los usaban para separar distintas secciones de código en "páginas".

3.5. Caracteres Unicode

Las barras diagonales inversas también se pueden usar para escribir caracteres Unicode arbitrarios. Se escriben como \u seguido del código del carácter Unicode (en hexadecimal).

Los códigos Unicode se aceptan sin importar que tengan mayúsculas o minúsculas.

```
print('\u00f1')
print('\u00F1')
```

Salida:

```
ñ
ñ
```

El sitio web de Unicode contiene más información sobre estos caracteres y sobre este estándar. Este sitio web tiene una tabla con los códigos.

3.6. Strings multilínea

Este es un tipo especial de string, que se escribe entre comillas triples ''' ''' o """ """, y que reconoce los saltos de línea sin necesidad de usar la secuencia \n.

```
print("""Esta es
una cadena de
caracteres
multilínea""")
```

Salida:

```
Esta es
una cadena de
caracteres
multilínea
```

3.7. Comentarios multilínea

Los comentarios multilínea no existen formalmente en Python, pero se puede hacer algo parecido usando strings multilínea fuera de lugares donde normalmente se usarían strings.

```
"""
Este no es un comentario real,
pero es ignorado por el
intérprete de Python
"""
```

La razón por la que esto ocurre es porque estos "comentarios" se usan para producir documentación del código (docstring). Si se ingresan en secciones particulares del código cumplirán su función de documentar, pero el intérprete no impide que se usen fuera de dichas secciones.

Se detallará mejor este uso en el capítulo de funciones.

3.8. Concatenación de strings

Dos o más cadenas se pueden unir una después de la otra, usando un proceso llamado concatenación. Se usa el operador +.

```
print("Hola" + " " + "mundo")
print("1" + "1")
```

Salida:

```
Hola mundo
11
```

La concatenación sólo se puede realizar entre strings, no entre cadenas y números.

```
print(1 + "1")
```

Salida:

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

3.9. Multiplicación de strings

Las cadenas también pueden ser multiplicadas por números enteros. Esto produce una versión repetida de la cadena original. El orden de la cadena y el número no importa, pero la cadena suele ir primero.

```
print("ja" * 4)
print(4 * "ja")
```

Salida:

```
jajajaja
jajajaja
```

También se pueden combinar operaciones de multiplicación y concatenación.

```
print(("j" + "a") * 4)  # jajajaja
```

Multiplicar por 0 genera un string vacío.

```
print("hola" * 0)
```

Salida:

```

```

3.10. Opciones del método print()

El método print() puede aceptar más de un string como argumento, lo cual hace que se muestren en una misma línea separados por espacios.

```
print('a', 'b', 'c')
```

Salida:

```
a b c
```

El método print() tiene 2 argumentos que pueden definirse para dar más control sobre lo que se imprime en la pantalla.

El argumento sep define el string separador entre cada string "normal" que se le entregue al método print(), excepto después del último. Estos separadores pueden incluir secuencias de escape.

```
print('a', 'b', 'c', sep='')
print('a', 'b', 'c', sep='.')
print('a', 'b', 'c', sep=', ')
print('a', 'b', 'c', sep='\t')
print('a', 'b', 'c', sep='\n')
```

Salida:

```
abc
a.b.c
a, b, c
a	b	c
a
b
c
```

El argumento end define el string que irá después del último string "normal". Se puede usar para seguir escribiendo en la misma línea, si no se incluye la secuencia de escape \n.

```
print('a', 'b', 'c', end=' ')
print('x')  # irá en la misma línea

print('a', 'b', 'c', end='\t')
print('x')  # irá en la misma línea
```

Salida:

```
a b c x
a b c	x
```

```
print('a', 'b', 'c', end='.\n')
print('x')  # irá en la línea siguiente
```

Salida:

```
a b c.
x
```

Ambos argumentos se pueden combinar.

```
print('[', end='')
print('a', 'b', 'c', sep=', ', end =']\n')
```

Salida:

```
[a, b, c]
```

Capítulo 4

Variables

4.1. Asignación de variables

Una variable permite almacenar un valor asignándole un nombre, el cual puede ser usado para referirse al valor más adelante en el programa. Para asignar una variable, se usa el signo igual =.

```
nombre = "Juan"
```

4.2. Nombre de variables válidos

Se aplican ciertas restricciones con respecto a los caracteres que se pueden usar en los nombres de variables. Los únicos caracteres permitidos son letras, números y guiones bajos. Además, no se puede comenzar con números o incluir espacios.

```
123 = 44
```

Salida:

```
SyntaxError: cannot assign to literal
```

```
123variable = 'hola'
```

Salida:

```
SyntaxError: invalid syntax
```

```
hola mundo = 'hola mundo'
```

Salida:

```
SyntaxError: invalid syntax
```

Python es sensible a mayúsculas y minúsculas, lo que significa que las variables “num”, “Num”, “NUM”, etc. son distintas.

```
num = 3
Num = 5
NUM = 10

print(NUM)
print(num)
print(Num)
```

Salida:

```
10
3
5
```

4.3. Palabras clave

Existen palabras específicas que tampoco se pueden usar como nombres de variables. El intérprete de Python reconoce estas palabras como palabras clave o keywords, y tienen usos reservados.

A continuación, se muestra una lista de todas las palabras clave en Python.

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

Nótese el uso de mayúsculas al principio de False, None y True.

4.4. Operaciones con variables

Se pueden usar variables dentro de operaciones. Lo único que se debe recordar es que deben declararse antes.

```
x = 6
y = 7
print(x + y)
```

Salida:

```
13
```

Una variable también puede cambiar de valor a lo largo de la ejecución de un programa.

```
x = 3
y = 4
z = x + y  # 7
z = z + 1  # 8
z = z * 2  # 16
```

4.5. Entrada

Para obtener información del usuario, se puede usar la función input(). La información obtenida puede ser almacenada como una variable.

```
x = input()
print(x)
```

Entrada:

```
5
```

Salida:

```
5
```

Toda la información recibida por el método input() es retornada como un string. También se puede entregar un string como parámetro al método input(), lo cual mostrará texto antes de pedir la entrada. Esto sirve para aclarar qué entrada está solicitando el programa.

```
nombre = input('Ingrese su nombre: ')
```

Al usar la función input(), el flujo del programa se detiene hasta que el usuario ingrese algún valor.

4.6. Conversión de tipos de datos

Existen funciones para convertir datos de un tipo a otro. Entre dichas funciones se encuentran:

- int(): Convierte a un número entero.

```
x = '7'

print(x)
print(int(x))
```

Salida:

```
7
7
```

Aunque ambos se impriman igual, el primero es un string y el segundo un entero (int).

- float(): Convierte a un float.

```python
num = 42

print(num)
print(float(num))
```

Salida:

```
42
42.0
```

- str(): Convierte a un string o cadena de caracteres. Uno de sus usos principales es para concatenar distintos tipos de datos.

```python
edad = 25
print("Edad: " + str(edad))
```

Salida:

```
Edad: 25
```

4.7. Operadores de asignación

Los operadores de asignación permiten escribir código como 'x = x + 1' de manera más concisa, como 'x += 1'. Lo mismo es posible con otros operadores como -, *, /, //, % y **.

```python
x = 10
x += 1  # 11
x -= 2  # 9
x *= 3  # 27
x /= 4  # 6.75
x //= 5  # 1.0
x %= 6  # 1.0
x **= 7  # 1.0
```

También se pueden usar con los operadores de concatenación y multiplicación de strings.

```python
x = "ja"  # ja
x += "ja"  # jaja
x *= 3  # jajajajajaja
```

Capítulo 5

Declaraciones if y lógica

5.1. Booleanos

Otro tipo de dato en Python es el tipo booleano, el cual sólo puede tener 2 valores: True y False. Estos se escriben empezando con mayúscula, al contrario de otros lenguajes de programación.

Se pueden crear directamente, aunque esto no tiene mucha utilidad en la mayoría de casos.

```
print(True)  # verdadero
print(False)  # falso
```

Salida:

```
True
False
```

Su uso más común es como el resultado obtenido al comparar valores, por ejemplo, usando el operador de igualdad ==.

```
print(0 == 0)
print(1 == 2)
print("hola" == 'hola')
```

Salida:

```
True
False
True
```

En Python, 1 y True significan lo mismo. Lo mismo ocurre con 0 y False.

```
print(1 == True)
print(0 == False)
```

Salida:

```
True
True
```

Usando otros términos, podría decirse que 1 es "truthy" y que 0 es "falsy".

5.2. Operadores de comparación

Los operadores de comparación usados en Python son los siguientes:

- Igual $=$, escrito como `==`: Se evalúa como True si ambos elementos son iguales.
- Distinto $\neq$, escrito como `!=`: Se evalúa como True si ambos elementos son distintos.
- Mayor que $>$, escrito como `>`: Se evalúa como True si el primer elemento es mayor.
- Menor que $<$, escrito como `<`: Se evalúa como True si el primer elemento es menor.
- Mayor o igual que $\geq$, escrito como `>=`: Se evalúa como True si el primer elemento es mayor o igual al segundo.
- Menor o igual que $\leq$, escrito como `<=`: Se evalúa como True si el primer elemento es menor o igual al segundo.

En los casos contrarios, los operadores retornan False.

Ejemplos de uso de operadores de comparación:

```
print(1 == 1)
print(4 != 4)
print(5 > 7)
print(-1 < 0)
print(3 >= 3)
print(0 <= -10)
```

Salida:

```
True
False
False
True
True
False
```

También se conocen como operadores relacionales.

Las comparaciones entre datos de tipo entero y float también son válidas.

```
print(1 == 1.0)  # True
```

Las comparaciones entre números y strings son válidas, pero siempre se evalúan como False.

```
print(1 == "1")  # False
print(1.0 == "1.0")  # False
print(1.0 == "1")  # False
```

5.3. Declaración if

Las declaraciones if sirven para ejecutar código si ser cumple una determinada condición. Si la expresión se evalúa como True, se lleva a cabo el bloque de código que le sigue. En caso contrario, no ocurre nada.

```
if 3 > 2:
    print('3 es mayor que 2')
```

Python usa indentación (espacio en blanco al comienzo de una línea) para delimitar bloques de código. En otros lenguajes, esta operación se realiza delimitando cada bloque de código por llaves .

5.4. Declaración if-else

Las declaraciones if-else son muy similares a las declaraciones if, pero esta vez sí ocurre algo cuando no se cumple la expresión a evaluar.

```
x = 5
y = 6
if x > y:
    print(str(x) + ' es mayor que ' + str(y))
else:
    print(str(y) + ' es mayor que ' + str(x))
```

```
x = input()
if x % 2 == 0:
    print(str(x) + ' es par')
else:
    print(str(x) + ' es impar')
```

5.5. Declaración elif

El término elif se usa cuando se encadenan múltiples declaraciones if-else. Hace que el código sea más corto.

```
x = input("Ingrese un número: ")
if x == 1:
    print("Uno")
elif x == 2:
    print("Dos")
elif x == 3:
    print("Tres")
else:
    print("Otro número")
```

El programa irá evaluando cada expresión una por una hasta que alguna se cumpla. Cuando esto ocurra, ejecutará las instrucciones en su bloque de código y continuará con las expresiones después del bloque de código else.

5.6. Operadores lógicos

Los operadores lógicos se utilizan para crear condiciones complejas agrupando condiciones más simples. Los operadores lógicos que se usan en Python son and, or y not.

El operador and se evalúa como True si ambas condiciones se cumplen.

```
print(1 == 1 and 2 == 2)  # True
print(5 > 3 and 3 > 4)  # False
```

El operador or se evalúa como True si al menos una de las condiciones se cumple.

```
print(1 == 1 or 2 == 2)  # True
print(5 > 3 or 3 > 4)  # True
```

El operador not sólo toma una condición y la invierte.

```
print(not 1 > 100)  # True
```

5.7. Precedencia de operadores lógicos

Los operadores lógicos siguen un orden similar a PEMDAS, el cual es el siguiente:

1. Paréntesis
2. Operador not
3. Operador and
4. Operador or

Sin embargo, este orden es confuso, por lo que se recomienda usar paréntesis para hacerlo más obvio.

```
print(not (False and False) or True)
# not (False and False) or True
# not False or True
# True

print(not (False and (True or False)))  # True
# not (False and (True or False))
# not (False and True)
```

```
# not False
# True
```

Capítulo 6

Listas

6.1. Creación de listas

Las listas se utilizan para almacenar varios elementos. Se crean utilizando corchetes y separando los elementos por comas.

```
palabras = ["hola", "mundo"]
```

En Python, las listas pueden tener elementos de más de un tipo.

```
elementos = ["Hola", 2, "mundo", 3.0, True]
```

6.2. Indexación de listas

Se puede acceder a un determinado elemento de la lista utilizando su índice entre corchetes. El índice del primer elemento es 0, en lugar de 1, como podría esperarse.

```
animales = ["Perro", "Gato"]
print(animales[0])  # Perro
print(animales[1])  # Gato
```

Alternativamente, se pueden usar índices negativos. El último elemento de una lista tiene índice -1, y va disminuyendo hasta llegar al primero.

```
animales = ["Perro", "Gato", "Mono", "León"]
print(animales[-1])  # León
print(animales[-2])  # Mono
print(animales[-3])  # Gato
print(animales[-4])  # Perro
```

6.3. IndexError en listas

Intentar acceder a un índice no existente producirá un error.

```
elementos = [False, 1, 2.0, "tres", [4]]
print(elementos[5])  # IndexError
```

6.4. Lista vacía

A veces es necesario crear una lista vacía y completarla más tarde durante la ejecución del programa. Las listas vacías se crean con un par de corchetes vacíos.

```
vacio = []
```

6.5. Anidación de listas

Las listas se pueden anidar dentro de otras listas, y para acceder a algún elemento se debe usar una cantidad mayor de corchetes, donde los índices indican el elemento desde afuera hacia adentro.

```
lista = ["hola", "mundo", [1, 2, 3]]
print(lista[0])  # hola
print(lista[1])  # mundo
print(lista[2])  # [1, 2, 3]
print(lista[2][0])  # 1
print(lista[2][1])  # 2
print(lista[2][2])  # 3
```

En el ejemplo de arriba se muestra como acceder a todos los elementos de la lista creada.

Las listas anidadas se pueden utilizar para representar cuadrículas 2D, como por ejemplo matrices.

```
identidad = [[1, 0, 0],
             [0, 1, 0],
             [0, 0, 1]]
```

6.6. Comas sobrantes

Se puede dejar a lo más una coma sobrante dentro de los corchetes de una lista. Este último elemento no será añadido a la lista.

```
letras = ['a', 'b', 'c',]
print(letras)  # ['a', 'b', 'c']
letras = ['a', 'b', 'c',,]  # SyntaxError
```

6.7. Operaciones con listas

El elemento en un índice determinado de una lista se puede reasignar.

```
numeros = [0, 0, 0, 0]
numeros[2] = 1
print(numeros)  # [0, 0, 1, 0]
```

Las listas se pueden agregar y multiplicar de la misma manera que las cadenas.

```
nums = [1, 2, 3]
print(nums + [4, 5, 6])  # [1, 2, 3, 4, 5, 6]
print(nums * 4)  # [1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
```

6.8. Operadores in y not in

Se puede verificar si un elemento pertenece a una lista usando el operador in. Este devuelve True si dicho elemento aparece una o más veces en la lista y False si no es así.

```
frutas = ["manzana", "naranja", "pomelo", "limón", "durazno", "damasco"]
print("manzana" in frutas)  # True
print("frutilla" in frutas)  # False
print("durazno" in frutas)  # True
```

Para comprobar si un elemento no pertenece a una lista, se pueden combinar los operadores in y not. Ambas formas de hacerlo son válidas.

```
frutas = ["manzana", "naranja", "pomelo", "limón", "durazno", "damasco"]
print("frutilla" not in frutas)  # True
print(not "frutilla" in frutas)  # True
```

6.9. Funciones de listas

Existen métodos que sirven para trabajar con listas.

- El método append() se usa para añadir un elemento al final de una lista.

```
lista = [1, 2, 3, 4, 5]
lista.append(6)
print(lista) # [1, 2, 3, 4, 5, 6]
```

- El método pop() elimina el último elemento de una lista.

```
lista = [1, 2, 3, 4, 5]
lista.pop()
print(lista)  # [1, 2, 3, 4]
```

- El método len() retorna el tamaño (número de elementos) de una lista.

```
letras = ['a', 'b', 'c', 'd', 'e', 'f']
print(len(letras))  # 6
```

- El método insert() se usa para insertar un elemento en un índice específico de una lista. Los elementos que están después del elemento insertado se desplazan a la derecha.

```
palabras = ["hola", "mundo"]
palabras.insert(1, "chao")
print(palabras)  # ['hola', 'chao', 'mundo']
```

- El método extend() se usa para añadir todos los elementos de una lista al final de otra lista.

```
nums1 = [1, 2, 3]
nums2 = [4, 5, 6]
nums1.extend(nums2)
print(nums1)  # [1, 2, 3, 4, 5, 6]
```

- El método index() encuentra la primera aparición de un elemento en una lista y devuelve su índice. Si el elemento no está en la lista, muestra un ValueError.

```
letras = ['a', 'z', 'x', 'b', 'd', 'h', 'k']
print(letras.index('x'))  # 2
print(letras.index('d'))  # 4
print(letras.index('m'))  # ValueError
```

- El método count() cuenta la cantidad de ocurrencias de un elemento en una lista.

```
lista = [1, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0]
print(lista.count(0))  # 7
print(lista.count(1))  # 5
```

- El método remove() elimina la primera ocurrencia de un elemento en una lista.

```
numeros = [1, 2, 3, 1, 2, 3]
numeros.remove(3)
print(numeros)  # [1, 2, 1, 2, 3]
numeros.remove(4)  # ValueError
```

Una forma más segura de usar remove() es la siguiente:

```
numeros = [1, 2, 3, 1, 2, 3]
if 4 in numeros:
    numeros.remove(4)
print(numeros) # [1, 2, 3, 1, 2, 3]
```

- Los métodos max() y min() devuelven el elemento de una lista con valor máximo o mínimo, respectivamente.

```
nums = [1, 10, 100, 5, 6, -11, -52, 78, 0, 25]
print(max(nums))  # 100
print(min(nums))  # -52
```

Si son strings, los compara por orden alfabético. Se considera como "mayor" a las palabras que se encuentran al final después de ordenarlas alfabéticamente.

```
palabras = ["hola", "manzana", "libro", "casa", "variable", "método"]
print(max(palabras))  # variable
print(min(palabras))  # casa
```

- El método sum() calcula la suma de los elementos de una lista.

```
lista = [5, 6, 10]
print(sum(lista))  # 21
```

- El método reverse() invierte los elementos de una lista.

```
nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
nums.reverse()
print(nums)  # [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

- El método sort() ordena alfabéticamente una lista de strings, por defecto se realiza ascendentemente. Para realizar el orden descendentemente, se debe entregar el parámetro reverse=True.

```
letras = ['z', 'x', 'a', 'i', 'j', 'y', 'd', 'f', 'h', 'm', 'c']
letras.sort()
print(letras)  # ['a', 'c', 'd', 'f', 'h', 'i', 'j', 'm', 'x', 'y', 'z']
letras.sort(reverse=True)
print(letras)  # ['z', 'y', 'x', 'm', 'j', 'i', 'h', 'f', 'd', 'c', 'a']
```

También funciona con listas compuestas de números.

```
numeros = [5, -100, 25, 37, -22, 1, 0, -1, 99, -99]
numeros.sort()
print(numeros)  # [-100, -99, -22, -1, 0, 1, 5, 25, 37, 99]
```

- El método clear() quita todos los elementos de una lista.

```
letras = ['a', 'b', 'c']
letras.clear()
print(letras)  # []
```

6.10. Copiar listas

Al intentar copiar listas sólo usando su nombre, en realidad se está haciendo una lista que hace referencia a la lista copiada. Esto significa que los cambios que reciba una lista también los recibirá la otra.

```
a = ['x', 'y']
b = a
b[0] = 'z'
print(b)  # ['z', 'y']
print(a)  # ['z', 'y']
```

Para evitar esto, se puede usar el método copy() para hacer una copia de una lista en una nueva instancia.

```
a = ['x', 'y']
b = a.copy()
b[0] = 'z'
print(b)  # ['z', 'y']
print(a)  # ['x', 'y']
```

6.11. Strings como listas

6.12. Indexación de strings

Algunos tipos de variables, como las cadenas, se pueden indexar como listas. La indexación de cadenas se comporta como si estuviera indexando una lista que contiene cada carácter de la cadena.

```
hola = "Hola mundo"
print(hola[0])  # H
print(hola[4])  #   (un espacio)
```

Se debe recordar que el espacio ″ ″ también es considerado como símbolo y tiene su índice dentro de la cadena.

6.13. Conversión de strings a listas

El método list() convierte una cadena de caracteres en una lista de caracteres.

```
frase = "Hola mundo"
lista = list(frase)
print(lista)  # ['H', 'o', 'l', 'a', ' ', 'm', 'u', 'n', 'd', 'o']
```

El método split() convierte una cadena de caracteres en una lista de palabras, separadas por espacios “ “.

```
frase = "Hola mundo"
lista = frase.split()
print(lista)  # ['Hola', 'mundo']
```

Al método split() también se le puede entregar un string separador. La lista será formada con los elementos entre cada aparición de ese string.

```
parrafo = "Lorem ipsum dolor sit amet, consectetur adipiscing elit..."
lista = parrafo.split()
print(lista)  # ['Lorem', 'ipsum', 'dolor', 'sit', 'amet,', 'consectetur',
↪  'adipiscing', 'elit...']
lista = parrafo.split(',')
print(lista)  # ['Lorem ipsum dolor sit amet', ' consectetur adipiscing
↪  elit...']
lista = parrafo.split('e')
print(lista)  # ['Lor', 'm ipsum dolor sit am', 't, cons', 'ct', 'tur
↪  adipiscing ', 'lit...']
```

Capítulo 7

Bucles

7.1. Bucles while

Un bucle while se usa para repetir un bloque de código varias veces, siempre que se cumpla cierta condición. Generalmente se usan con una variable como contador.

```
i = 1  # contador
while i <= 5:
    print(i)
    i = i + 1
# muestra los números del 1 al 5
```

El código en el cuerpo de un bucle while se ejecuta repetidamente. Esto se llama iteración.

7.2. Bucles infinitos

Si la condición a evaluar es siempre verdadera, el bucle se ejecutará indefinidamente. Esto se llama bucle infinito.

```
i = 1
while True:
    print(i)
    i = i + 1
```

No se recomienda ejecutar este código. Si por algún motivo se ejecuta, se puede interrumpir su ejecución enviando un KeyboardInterrupt al usar la combinación de teclas Ctrl+C.

7.3. Declaración break

Para finalizar un bucle while prematuramente, se puede usar la declaración break.

```
i = 0
while True:
```

```
    print(i)
    i = i + 1

    if i >= 5:
        print("fin")
        break
# muestra los números del 0 al 4, seguido de "fin"
```

El uso de la declaración break fuera de un bucle provoca un error.

```
break  # SyntaxError
```

7.4. Declaración continue

Otra declaración que se puede utilizar dentro de los bucles es continue. A diferencia de break, continue salta de nuevo a la parte superior del bucle, en lugar de detenerlo. Básicamente, la declaración continue detiene la iteración actual y continúa con la siguiente.

```
print("Números del 1 al 10, exceptuando el 5:")
i = 0
while i < 10:
    i = i + 1
    if i == 5:
        continue
    print(i)
```

Al igual que la declaración break, usar continue fuera de un bucle provoca un error.

```
continue  # SyntaxError
```

7.5. Bucle for con listas

Los bucles for se pueden usar para iterar sobre una lista. Las instrucciones dentro de su bloque de código se ejecutarán para cada elemento de la lista.

```
animales = ["perro", "gato", "jirafa", "elefante", "mono"]
for animal in animales:
    print(animal)
# muestra los animales dentro de la lista
```

También se puede usar para iterar sobre cadenas.

```
texto = "probando bucles for"
for caracter in texto:
    print(caracter)
# muestra todos los caracteres del texto
```

De manera similar a los bucles while, las declaraciones break y continue se pueden utilizar en los bucles for para detener el bucle o saltar a la siguiente iteración.

```
numeros = [1, 9, 8, 2, 1, 0, 0, 2, 4, 0, 3, 5, 6, 8]
for n in numeros:
    if n == 0:
        continue
    if n == 3:
        break
    print(n)
# muestra los números de la lista, omitiendo los 0 y termina cuando llega al 3
```

7.6. Rangos

La función range() devuelve una secuencia de números. De forma predeterminada, comienza desde 0, se incrementa en 1 y se detiene antes del número especificado.

Si se llama a range() con un argumento, produce un objeto con valores desde 0 a el número antes de ese argumento.

```
numeros = list(range(10))
print(numeros)  # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Para mostrar los números dentro de un objeto range, debe convertirse a una lista.

Si se llama con 2 argumentos, produce valores desde el primero hasta antes del segundo.

```
numeros = list(range(3, 8))
print(numeros)  # [3, 4, 5, 6, 7]
```

```
print(range(10) == range(0, 10))  # True
```

El rango puede tener un tercer argumento, que determina el intervalo de la secuencia producida, también llamado como paso. Este número puede ser positivo (incremento) o negativo (decremento).

```
nums1 = list(range(0, 15, 2))
nums2 = list(range(15, 0, -2))
```

```
print(nums1)  # [0, 2, 4, 6, 8, 10, 12, 14]
print(nums2)  # [15, 13, 11, 9, 7, 5, 3, 1]
```

7.7. Bucle for en rangos

Una forma común de usar los bucles for es iterando sobre rangos. Esto permite hacer bucles de muchas formas.

```
for i in range(10):
    print(i)
# muestra los números del 0 al 9
```

No es necesario llamar list() en el objeto de rango cuando se usa en un bucle for, porque no se está indexando, por lo que no requiere una lista.

Capítulo 8

Funciones

8.1. ¿Qué es una función?

Cualquier sentencia que consista de una palabra seguida de información entre paréntesis es llamada una función.

Ejemplos de funciones que se han visto anteriormente.

```
print("Hola mundo")
x = input()
y = int("7")
z = float(3)
texto = str(False)
rango = range(100, 0, -1)
```

8.2. Definición de funciones

Para definir una función, se debe usar la palabra clave def, seguida del nombre de la función y de un paréntesis que puede o no incluir parámetros. El cuerpo de la función incluye el código y debe tener un grado de indentación mayor que el de la definición.

Algunos ejemplos de funciones:

```
def saludar():
    print("Hola")
```

```
def exclamar(palabra):
    print("!" + palabra + "!")
```

Una función puede pedir más de un argumento, los cuales deben separarse por comas.

```
def sumar(num1, num2):
    print(num1 + num2)
```

```
def mayor(num1, num2):
    if num1 >= num2:
        print(num1)
    else:
        print(num2)
```

Los argumentos de funciones pueden ser utilizados como variables dentro de la definición de la función. Sin embargo, no pueden ser referenciados fuera de la definición de la función. Esto también se aplica a las demás variables creadas dentro de una función.

8.3. Llamado de funciones

Para llamar una función, debe escribirse su nombre, seguido de un paréntesis que contiene los argumentos (información) que se le quieren entregar.

Ejemplos de uso de las funciones definidas anteriormente:

```
saludar()
exclamar("Hola mundo")
sumar(3, 2)
mayor(25, 50)
```

8.4. Devolución de valores en una función

Ciertas funciones devuelven un valor para ser utilizado más adelante. Para hacer esto en la definición de funciones nuevas, se debe usar la palabra clave return.

```
def sumar(x, y):
    return x + y

resultado = sumar(4, 5)
print(resultado)  # 9
```

Cualquier código luego de la sentencia return nunca ocurrirá.

```
def sumar(x, y):
    print('Esto si ocurrirá')
    return x + y
    print('Esto nunca ocurrirá')
```

```
5
6 print(sumar(4, 5))
```

8.5. Docstring

Las docstring (cadenas de documentación) cumplen un propósito similar al de los comentarios, pero son más específicos y tienen una sintaxis distinta.

Son creados colocando una cadena multilínea que contenga una explicación de la función por debajo de la primera línea de la función.

```
1 def sumar(x, y):
2     """
3     Retorna la suma de los 2 números ingresados
4     """
5     return x + y
```

A diferencia de los comentarios convencionales, los docstring se conservan a lo largo del tiempo de ejecución del programa. Esto le permite al programado examinar estos comentarios en el tiempo de ejecución.

Una forma de accederlo es usando la variable __doc__.

```
1 print(print.__doc__)
```

Otra forma es usando la función help(). Esto entrega un poco más de información.

```
1 help(print)
```

Normalmente se usan docstrings para explicar los parámetros y lo que retorna una función.

```
1 def sumar(x, y):
2     """
3     Parámetros:
4     x: Primer número
5     y Segundo número
6     Retorna:
7     La suma de ambos números
8     """
9     return x + y
```

8.6. Funciones como objetos

Aunque sean creadas de manera diferente que las variables regulares, las funciones son como cualquier otra clase de valor. Pueden ser asignadas y reasignadas a variables, y luego ser referenciadas por esos nombres.

```
def multiplicar(x, y):
    return x * y

a = 4
b = 7
producto = multiplicar

print(multiplicar(a, b))  # 28
print(producto(a, b))  # 28
```

Las funciones también pueden ser usadas como argumentos de otras funciones.

```
def sumar(x, y):
    return x + y

def repetir2veces(funcion, x, y):
    return funcion(funcion(x, y), funcion(x, y))

a = 5
b = 10

print(repetir2veces(sumar, a, b))
# sumar(sumar(5, 10), sumar(5, 10))
# sumar(5 + 10, 5 + 10)
# sumar(15, 15)
# 15 + 15
# 30
```

8.7. Sobrecarga de funciones

En Python, la sobrecarga de funciones se define como la habilidad de que una función se comporte de distinta manera dependiendo del número de argumentos que reciba. Esto permite reutilizar el mismo código, reducir su complejidad y hacer que sea más fácil de leer.

Para sobrecargar una función, se deben definir los parámetros que sean opcionales con un valor por defecto None. Después, se usan declaraciones if-elif-else para revisar todos los casos posibles.

```
def saludar(nombre=None):
    if nombre is not None:
        print("Hola, " + nombre)
    else:
        print("Hola")

saludar()  # Hola
saludar("Juan")  # Hola, Juan
```

```python
def area(x, y=None):
    if y is None:  # Cuadrado
        return x * x
    else:  # Rectángulo
        return x * y

area(4)  # 16, cuadrado
area(5, 6)  # 30, rectángulo
```

8.8. Anotaciones de tipos

Una anotación de tipos es, como su nombre lo dice, una notación opcional que especifica el tipo de los parámetros de una función y su tipo de retorno.

```python
def duplicar(mensaje: str) -> str:
    return mensaje + mensaje

resultado = duplicar("hola")
print(resultado)  # holahola
```

En el ejemplo de arriba, se muestra que mensaje es un parámetro de tipo string y que la función duplicar() retorna un string.

Esto le permite al programador saber qué tipos de datos se le deben entregar una función y que tipos de datos esperar cuando esta función retorne.

Las anotaciones de tipos son ignoradas completamente por el intérprete de Python. No restringen el tipo de los parámetros o del retorno de una función, pero son muy útiles al momento de documentar.

```python
def sumar(x: int, y: int) -> int:
    return x + y

# Uso esperado:
print(sumar(10, 5))  # 15

# Usos válidos pero no esperados:
print(sumar(10.2, 7.4))  # 17.6
print(sumar('a', 'b'))  # ab
```

Aunque el intérprete de Python los ignore, existen algunos IDEs y programas que pueden analizar código que contiene anotaciones de tipos y alertar sobre problemas potenciales.

8.9. El objeto None

El objeto None es utilizado para representar la ausencia de un valor. Es similar a null en otros lenguajes de programación.

Al igual que otros valores “vacíos”, tales como (), [], {} y la cadena vacía ″″, es False cuando es convertido a una variable booleana.

Cuando es ingresado a la consola de Python, se visualiza como una cadena vacía.

```
None  #
print(None)  # None
None == None  # True
```

El objeto None es devuelto por cualquier función que no devuelve explícitamente algo más.

```
def saludar():
    print("Hola mundo")

var = saludar()
print(var)  # None
```

Capítulo 9

Módulos y la biblioteca estándar

9.1. Módulos

Los módulos son pedazos de código que otras personas han escrito para cumplir tareas comunes tales como generar números aleatorios, realizar operaciones matemáticas, etc.

La manera básica de utilizar un módulo es agregar una declaración import en la parte superior del código, y luego usar su nombre para acceder a las funciones y variables dentro del módulo.

```
import random

for i in range(5):
    valor = random.randint(1, 6)
    print(valor)
# muestra 5 números generados aleatoriamente
```

Este ejemplo importa el módulo random y usa su función randint() para generar 5 números aleatorios en el rango del 1 al 6.

Hay otra clase de import que puede ser utilizada si sólo necesitas ciertas funciones de un módulo.

```
from math import pi

print(pi)  # 3.141592653589793
```

Para importar más de un elemento, se debe hacer una lista separada por comas.

```
from math import pi, e, sqrt, sin, cos
```

9.2. Error de importación

Importar un módulo que no está disponible dará un ImportError.

```
import modulo_desconocido
```

9.3. Alias

Se puede importar un módulo u objeto bajo un nombre distinto utilizando la palabra clave as y entregándole un alias. Esto se usa principalmente cuando un módulo u objeto tiene un nombre largo o confuso.

```
from math import sqrt as raiz_cuadrada

print(raiz_cuadrada(42))  # 6.48074069840786
```

También se puede usar un alias con más de un objeto.

```
from math import sqrt as raiz_cuadrada, pi as numero

print(raiz_cuadrada(numero))  # 1.7724538509055159
```

9.4. La biblioteca estándar

Hay 3 tipos principales de módulos en Python: aquellos que escribes tú mismo, aquellos que se instalan de fuentes externas y aquellos que vienen preinstalados con Python.

El último tipo se denomina la biblioteca estándar, y contiene muchos módulos útiles. Algunos de estos módulos son:

string	random	socket	unittest
re	os	email	pdb
datetime	multiprocessing	json	argparse
math	subprocess	doctest	sys

La extensa biblioteca estándar de Python es una de sus principales fortalezas como lenguaje. Se puede encontrar más información sobre los módulos de la biblioteca estándar en la documentación.

Algunos de los módulos en la biblioteca estándar están escritos en Python y otros en C. La mayoría están disponibles en todas las plataformas, pero algunos son específicos de Windows o Unix.

9.5. Módulos externos y pip

Muchos módulos de Python creados por terceros son almacenados en el índice de paquetes Python (Python Package Index, PyPI). Se puede ver el repositorio en su sitio web oficial.

La mejor manera de instalar estos es utilizando un programa llamado pip. Este viene instalado por defecto con las distribuciones modernas de Python.

Para instalar una biblioteca, se debe buscar su nombre, ir a la línea de comandos y escribir pip install nombre.

```
pip install nombre_de_libreria
```

Es importante recordar que los comandos de pip se deben introducir en la línea de comandos, no en el interpretador de Python.

Se puede ingresar el comando pip help para ver información sobre otros comandos que se pueden usar con este gestor de paquetes.

```
pip help
```

Utilizar pip es la forma estándar de instalar bibliotecas en la mayoría de sistemas operativos, pero algunas bibliotecas tienen binarios predefinidos para Windows. Estos son archivos ejecutables regulares que permiten instalar bibliotecas con una interfaz gráfica de la misma manera que se instalan otros programas.

Capítulo 10

El módulo math

Capítulo 11

El módulo random

Capítulo 12

Excepciones

12.1. Excepciones

Se han mencionado excepciones en las secciones anteriores. Ocurren cuando algo sale, mal, debido a código incorrecto o entradas incorrectas. Cuando ocurre una excepción, el programa se detiene inmediatamente.

Al intentar dividir por 0, se produce un ZeroDivisionError.

```
1 print(10 / 0)
```

Otras excepciones comunes son:

- ImportError: Cuando falla una importación.
- IndexError: Cuando se intenta indexar una lista con un número fuera de rango.
- NameError: Cuando una variable desconocida es utilizada.
- SyntaxError: Cuando el código no puede ser analizado correctamente.
- TypeError: Cuando una función es llamada con un valor de tipo inapropiado
- ValueError: Cuando una función es llamada con un valor del tipo correcto, pero con un valor incorrecto.

Las bibliotecas creadas por terceros a menudo definen sus propias excepciones.

12.2. Declaración try-except

Para manejar excepciones y ejecutar código cuando ocurre una excepción, se puede usar una sentencia try-except. El bloque try contiene código que puede lanzar una excepción. Si ocurre una excepción, el código en el bloque try deja de ser ejecutado y el código en el bloque except se ejecuta. Si no ocurre ningún error, el código en el bloque except no se ejecuta.

```
1 x = int(input())
2 y = int(input())
3
4 try:
```

```
    print("Resultado de la división: " + str(x / y))
except ZeroDivisionError:
    print("Error: Se intentó dividir por 0")
```

Una sentencia try puede tener varios bloques except para manejar diferentes excepciones. Varias excepciones pueden ser colocadas dentro de un mismo bloque except utilizando paréntesis.

```
x = input()
y = input()

try:
    print("Resultado de la división: " + str(x / y))
except ZeroDivisionError:
    print("Error: Se intentó dividir por 0")
except (ValueError, TypeError):
    print("Error: Valor o tipo no válido")
```

Una sentencia except sin ninguna excepción especificada atrapa todos los errores. Estos deben usarse con moderación ya que pueden atrapar errores inesperados y esconder errores de programación.

```
palabra = 'spam'
numero = 0

try:
    print(palabra / numero)
except:
    print("Ha ocurrido un error")
```

Si una sentencia except vacía viene acompañada de otros except, esta debe ir al final, para evitar atrapar errores esperados.

12.3. Declaración finally

Para asegurar que algún código se ejecute sin importar cuál error ocurra, se puede usar la palabra clave finally. La sentencia finally se coloca en el fondo de una sentencia try-except. El código dentro de la sentencia finally siempre se ejecuta después del código del bloque try y de cualquier bloque except que se ejecute.

```
try:
    print("Hola")
    print(1 / 0)
except:
    print("Error: División por 0")
finally:
    print("Este mensaje siempre se mostrará")
```

El código dentro del bloque finally se ejecutará incluso si una excepción sin atrapar ocurre en alguno de los bloques que lo preceden.

```
try:
    print(1)
    print(1 / 0)
except ZeroDivisionError:
    print(variable_desconocida)
finally:
    print("Este mensaje siempre se mostrará")

print("Este mensaje no se mostrará")
```

12.4. Levantar excepciones

Se puede usar la sentencia raise para levantar excepciones. Se necesita especificar el tipo de la excepción levantada.

```
print(1)
raise ValueError
print(2)
```

Las excepciones pueden ser levantadas con argumentos que den detalles sobre ellas.

```
nombre = "123"
raise NameError("¡Nombre no válido!")
```

```
password = input("Ingresa tu contraseña: ")

if len(password < 8):
    raise ValueError("La contraseña debe tener 8 o más caracteres")
```

En los bloques except, la sentencia raise puede ser utilizada sin argumentos para volver a levantar cualquier excepción que haya ocurrido.

```
try:
    print(7 / 0)
except:
    print("Ha ocurrido un error")

raise
```

12.5. Aserciones

Una aserción es una comprobación de validez, la cual prueba una expresión. La expresión es probada, y si el resultado es falso, usa excepción AssertionError es levantada.

Las aserciones son llevadas a cabo a través de la declaración assert.

```
print('a')
assert 1 == 1  # True
print('b')
assert 1 > 10  # False
print('c')  # AssertionError
```

Los programadores a menudo colocan aserciones al principio de una función para asegurarse de que la entrada sea válida, y luego de llamar una función para revisar la validez de la salida.

La declaración assert puede recibir un segundo argumento el cual es pasado a la excepción AssertionError levantada si la aserción falla.

```
print('a')
assert 1 == 1  # True
print('b')
assert 1 > 10  # False
print('c')  # AssertionError
```

Las excepciones AssertionError pueden ser atrapadas y manejadas como cualquier otra excepción utilizando la sentencia try-except, pero si no son manejadas, este tipo de excepción terminará el programa.

En general, se recomienda usar aserciones para atrapar tus propios errores y excepciones para atrapar los errores que los usuarios u otras personas podrían cometer.

```
assert 1 == 100, "Ha ocurrido un error"
```

Capítulo 13

Pruebas unitarias

Capítulo 14

Archivos

14.1. Abrir archivos

Python se puede usar para leer y escribir los contenidos de archivos. Los archivos de texto son los más fáciles de manipular. Antes de que un archivo pueda ser editado, debe ser abierto con la función open().

```
archivo = open("archivo.txt")
```

El argumento de la función open es la ruta del archivo. Si el archivo se encuentra en el directorio hábil actual del programa, se puede especificar sólo su nombre (ruta relativa).

14.2. Modos de apertura

Se puede especificar el modo utilizado para abrir un archivo al pasar un segundo argumento a la función open().

Los modos de apertura son:

- "r": Significa modo de lectura, el cual es el modo predeterminado.

```
# Modo de lectura
archivo = open("archivo.txt", "r")
archivo = open("archivo.txt")
```

- "w": Significa modo de escritura, el cual sirve para reescribir los contenidos de un archivo. Abrir un archivo en este modo inmediatamente borra todos sus contenidos.

```
# Modo de escritura
archivo = open("archivo.txt", "w")
```

- "x": Significa modo de creación, se usa para crear un fichero y escribir sobre él. Entrega un error FileExistsError si el archivo ya existe o no está vacío.

```python
# Modo de creación
archivo = open("archivo.txt", "x")
```

- "a": significa modo de anexo, para agregar nuevo contenido al final de un archivo.

```python
# Modo de anexo
archivo = open("archivo.txt", "a")
```

Además de los modos de apertura, también existen los modos en los que se muestra la información:

- "t": significa abrir el archivo en modo texto (predeterminado). No es necesario escribirlo.
- "b": significa abrir el archivo en modo binario, que es utilizado para archivos que no son de texto (tales como archivos de imágenes o sonido). Se debe combinar con alguno de los modos anteriores.

```python
# Modo de lectura en binario
archivo = open("archivo.txt", "rb")

# Modo de escritura en binario
archivo = open("archivo.txt", "wb")

# Modo de anexo en binario
archivo = open("archivo.txt", "ab")

# Modo de creación en binario
archivo = open("archivo.txt", "xb")
```

14.3. Extensión de modos de apertura

Se puede añadir el signo "+" a cualquiera de los modos de arriba para darles acceso adicional a archivos.

La siguiente tabla muestra el funcionamiento de cada modo:

	r	r+	w	w+	x	x+	a	a+
Lee el archivo	Sí	Sí	No	Sí	No	Sí	No	Sí
Escribe en el archivo	No	Sí	Sí	Sí	Sí	Sí	Sí	Sí
Crea el archivo si no existe	No	No	Sí	Sí	Sí	Sí	Sí	Sí
Borra todos los contenidos del archivo	No	No	Sí	Sí	No*	No*	No	No
Posición del cursor	Inicio	Inicio	Inicio	Inicio	Inicio	Inicio	Final	Final

*: Levanta una excepción FileExistsError.

14.4. Cierre de archivos

Una vez que un archivo haya sido abierto y utilizado, es un buen hábito cerrarlo. Esto se logra con el método close() de un objeto archivo.

```
archivo = open("archivo.txt", "w")

# Trabajar con el archivo...

archivo.close()
```

A nivel de sistemas operativos, cada proceso tiene un límite en la cantidad de archivos que puede abrir simultáneamente. Si un programa se ejecuta por mucho tiempo y podría abrir muchos archivos, se debe tener cuidado.

14.5. Lectura de archivos

Los contenidos de un archivo que ha sido abierto en modo texto pueden ser leídos utilizando el método read().

```
archivo = open("archivo.txt", "r")

contenidos = archivo.read()
archivo.close()  # Ya se almacenaron los contenidos del archivo

print(contenidos)
```

Para leer sólo una determinada parte de un archivo, se puede proveer un número como argumento a la función read(). Esto determina el número de bytes que deberían ser leídos.

Se pueden hacer más llamadas a read() en el mismo objeto archivo para leer más de él byte por byte. Si no se le pasan argumentos, o si el argumento es negativo, read() devuelve el resto del archivo.

```
archivo = open("archivo.txt", "r")

print(archivo.read(16))
print(archivo.read(8))
print(archivo.read(4))
print(archivo.read())

archivo.close()
```

El ejemplo de arriba primero mostrará los bytes del 1 al 16, después del 17 al 24, del 25 al 28, y del 29 al último byte.

Luego de que todos los contenidos de un archivo hayan sido leídos, cualquier intento de leer más de ese archivo devolverá una cadena vacía "" porque se está intentando leer desde el final del archivo.

```
archivo = open("archivo.txt", "r")

print(archivo.read())
print(archivo.read())  # Cadena vacía
```

```
5
6 archivo.close()
```

Usando el método seek(), se puede mover el “cursor” (puntero) al principio del archivo.

```
1 archivo = open("archivo.txt", "r")
2
3 print(archivo.read())
4 archivo.seek(0)  # Regresa a la posición 0
5 print(archivo.read())
6
7 archivo.close()
```

El método tell() entrega la posición actual del puntero.

```
1 archivo = open("archivo.txt", "r")
2
3 print(archivo.read(16))
4 print(archivo.tell())
5
6 archivo.close()
```

Para obtener cada línea de un archivo, se puede usar la función readlines() para devolver una lista donde cada elemento es una línea del archivo.

```
1 archivo = open("archivo.txt", "r")
2
3 print(archivo.readlines())
4
5 archivo.close()
```

Esto también se puede hacer usando un bucle for.

```
1 archivo = open("archivo.txt", "r")
2
3 for linea in archivo:
4     print(linea)
5
6 archivo.close()
```

14.6. Escritura de archivos

Para escribir sobre archivos se utiliza el método write(), el cual escribe una cadena en un archivo.

```
# Escribe sobre el archivo
archivo = open("archivo.txt", "w")
archivo.write("¡Hola mundo!")
archivo.close()

# Muestra el texto del archivo
archivo = open("archivo.txt", "r")
print(archivo.read())
archivo.close()
```

Cuando un archivo es abierto en modo de escritura, el contenido existente del archivo es borrado.

```
# Contenidos iniciales del archivo
archivo = open("archivo.txt", "r")
print(archivo.read())
archivo.close()

# Se borran los contenidos y se escribe sobre ellos
archivo = open("archivo.txt", "w")
archivo.write("¡Hola mundo!")
archivo.close()

# Contenidos nuevos
archivo = open("archivo.txt", "r")
print(archivo.read())
archivo.close()
```

Para evitar que esto ocurra, se puede usar el modo anexo “a”.

```
archivo = open("archivo.txt", "a")

archivo.write("¡Hola mundo!")

archivo.close()
```

El método write() devuelve el número de bytes escritos en un archivo, si su llamada es exitosa.

```
archivo = open("archivo.txt", "w")

bytes_escritos = archivo.write("¡Hola mundo!")
print(bytes_escritos)

archivo.close()
```

Para escribir algo que no sea un string, necesita ser convertido primero a un string.

```
archivo = open("archivo.txt", "w")

archivo.write(str(42))

archivo.close()
```

14.7. Declaración with

Es buena práctica evitar gastar recursos asegurándose de que los archivos sean siempre cerrados después de utilizarlos. Una forma de hacer esto es utilizando try-finally.

```
try:
    archivo = open("archivo.txt")
    print(archivo.read())
except FileNotFoundError:
    print("Archivo no encontrado")
finally:
    try:
        archivo.close()
    except:
        print("No se puede borrar un archivo que no existe")
```

Esto asegura que el archivo sea siempre cerrado incluso si ocurre un error. Sin embargo, existe una forma más cómoda de hacer esto usando declaraciones with.

Una forma alternativa de trabajar con archivos es utilizando declaraciones with. Esto crea una variable temporal (a menudo llamada f), la cual solo es accesible en el bloque indentado de la declaración with.

```
with open("archivo.txt") as f:
    print(f.read())
```

El archivo se cierra automáticamente al final de la declaración with, incluso si ocurren excepciones dentro de ella.

Comparado con la forma tradicional de trabajar con archivos (open-close), usar with tiene el inconveniente de que los archivos tienen que volverse a abrir cada vez que se quiera trabajar con ellos.

Así, una forma más segura de trabajar con archivos puede tener la siguiente estructura:

```
try:
    with open("archivo.txt") as f:
        print(f.read())
except:
    print("Error")
```

Capítulo 15

Módulos time y datetime

Capítulo 16

Estructuras de datos

16.1. Estructuras de datos

16.2. Diccionarios

Los diccionarios son estructuras de datos utilizadas para mapear claves arbitrarias a valores. Pueden ser indexados de la misma manera que las listas, utilizando corchetes que contengan claves.

```
edades = {"Juan": 25, "Paula": 36, "John": 42, "Diego": 58}

print(edades["Juan"])
print(edades["John"])
```

Cada elemento de un diccionario es representado por un par clave:valor (key:value). Estos elementos no se almacenan con algún orden en específico. Los diccionarios no están ordenados, lo que significa que no pueden ser indexados por índices, sólo por claves.

Las listas pueden ser consideradas como diccionarios con claves de números enteros dentro de un cierto rango.

```
instrumentos = {0: "Violín", 1: "Piano", 2: "Guitarra", 3: "Batería"}

for i in range(0, 4):
    print(instrumentos[i])
```

Un diccionario puede almacenar como valor cualquier tipo de datos.

```
primarios = {
    "rojo": [255, 0, 0],
    "verde": [0, 255, 0],
    "azul": [0, 0, 255]
}

print(primarios["verde"])  # [0, 255, 0]
```

Tratar de indexar una clave que no es parte de un diccionario retorna un KeyError. Un diccionario vacío es definido como .

```
diccionario_vacio = {}
```

Sólo objetos inmutables pueden ser utilizados como claves de diccionario. Los objetos inmutables son aquellos que no pueden ser cambiados. Algunos objetos mutables son listas, conjuntos y diccionarios. Tratar de utilizar un objeto mutable como clave de diccionario ocasiona un TypeError.

16.3. Indexación de diccionarios

Al igual que las listas, las claves de un diccionario pueden ser asignadas a distintos valores. Sin embargo, a diferencia de las listas, se le puede asignar un valor a nuevas claves, no sólo a las que ya existen.

```
cuadrados = {1: 1, 2: 4, 3: "nueve", 4: 16}

cuadrados[6] = 36
cuadrados[3] = 9
print(cuadrados)  # {1: 1, 2: 4, 3: 9, 4: 16, 6: 36}
```

16.4. Uso de in y not en diccionarios

Para determinar si una clave está en un diccionario, se puede usar los operadores in y not in, al igual que en listas.

```
numeros = {
    1: "uno",
    2: "dos",
    3: "tres"
}

print(1 in numeros)  # True
print("tres" in numeros)  # False
print(4 not in numeros)  # True
```

Nótese que retorna False al buscar “tres”. Esto ocurre porque “tres” es un valor, no una clave.

16.5. Función get()

Un método útil de diccionarios es get(). Hace lo mismo que indexar, pero si una clave no es encontrada en el diccionario entonces devuelve otro valor especificado (‘None’ por defecto).

```
pares = {
    1: "manzana",
```

```
    "naranja": [2, 3, 4],
    True: False,
    None: "True"
}

print(pares.get("naranja"))  # [2, 3, 4]
print(pares.get(42))  # None
print(pares.get(12345, "no encontrado"))  # no encontrado
```

En este caso es importante recordar que 1 y True son la misma clave. Entonces, el diccionario "pares" sólo tiene 3 elementos (la clave True tendrá como valor False, su último valor).

```
print(pares)  # {1: False, 'naranja': [2, 3, 4], None: 'True'}
```

16.6. Función keys()

16.7. Tuplas

Las tuplas son estructuras de datos muy parecidas a las listas, excepto que son inmutables (no pueden ser cambiadas). También se crean utilizando paréntesis en vez de corchetes.

```
tupla = ("hola", "chao", "mundo")
```

Se puede acceder a los valores de una tupla utilizando su índice. Funciona de la misma forma que con listas.

```
vehiculos = ("Auto", "Moto", "Avión", "Barco", "Tren")

print(vehiculos[2])  # Avión
print(vehiculos[4])  # Tren
```

Tratar de reasignar un valor a una tupla ocasiona un TypeError.

```
animales = ("mono", "elefante")

animales[1] = "gato"  # TypeError
```

Al igual que las listas y diccionarios, las tuplas pueden ser anidadas entre sí. Las tuplas son inmutables, pero el contenido de elementos mutables dentro de ellas puede ser cambiado.

```python
tupla = (1, "hola", ['a', 'b'])
tupla[2][0] = 'x'
print(tupla)  # (1, 'hola', ['x', 'b'])
```

Las tuplas pueden "empaquetarse" o "desempaquetarse", lo que puede ser útil al momento de crear variables.

```python
frutas = ('manzana', 'naranja', 'frutilla')
f1, f2, f3 = frutas

print(f1)  # manzana
print(f2)  # naranja
print(f3)  # frutilla
```

Las tuplas pueden ser creadas sin paréntesis, simplemente separando los valores por comas.

```python
numeros = "uno", "dos", "tres"
print(numeros)  # ('uno', 'dos', 'tres')
```

Una tupla vacía se crea utilizando un par de paréntesis vacíos.

```python
tupla_vacia = ()
```

Las tuplas son más rápidas que las listas, pero no pueden ser modificadas.

16.8. Conjuntos

Los conjuntos son estructuras de datos parecidas a las listas o a los diccionarios. Son creandos utilizando llaves o la función set(). Comparten algunas de las funcionalidades de las listas, como el uso de in para revisar si contienen o no un elemento en particular.

```python
numeros = {1, 2, 3, 4, 5}
lenguajes = set(["Python", "Java", "C", "C++"])

print(3 in numeros)  # True
print("Python" not in lenguajes)  # False
```

Para crear un conjunto vacío, se debe utilizar set(), ya que crea un diccionario vacío.

```python
conjunto_vacio = set()
```

Los conjuntos difieren de las listas de varias formas, pero comparten varias operaciones de listas como len().

No están ordenados, lo cual significa que no pueden ser indexados. No pueden tener elementos duplicados.

Debido a la forma en que son almacenados, es más rápido revisar si un elemento es parte de un conjunto que si es parte de una lista.

En lugar de utilizar append() para agregarle algo al conjunto, se utiliza add(). El método remove() elimina un elemento específico de un conjunto.

```
nums = {1, 2, 1, 3, 1, 4, 5, 6}
print(nums)  # {1, 2, 3, 4, 5, 6}

nums.add(-7)
nums.remove(3)
print(nums)  # {1, 2, 4, 5, 6, -7}
```

El método pop() elimina un elemento arbitrario. Esto significa que debido a la forma en la Python implementa conjuntos, no hay garantía de que los elementos se retornarán en el mismo orden que en el que se añadieron.

Generalmente, elimina el primer elemento, pero esto no se puede garantizar.

```
nums = {1, 2, 3, 4, 5, 6}
nums.pop()
print(nums)
```

Usos básicos de conjuntos incluyen pruebas de membresía y la eliminación de entradas duplicadas.

16.9. Operaciones con conjuntos

Los conjuntos pueden ser combinados utilizando operaciones matemáticas.

El operador de unión — combina dos conjuntos para formar uno nuevo que contiene los elementos de cualquiera de los dos.

```
primero = {1, 2, 3, 4, 5, 6}
segundo = {4, 5, 6, 7, 8, 9}

# Unión
print(primero | segundo)  # {1, 2, 3, 4, 5, 6, 7, 8, 9}
```

El operador de intersección & obtiene sólo los elementos que están en ambos.

```
primero = {1, 2, 3, 4, 5, 6}
segundo = {4, 5, 6, 7, 8, 9}

# Intersección
print(primero & segundo)  # {4, 5, 6}
```

El operador de diferencia - obtiene los elementos que están en el primer conjunto, pero no en el segundo.

```
primero = {1, 2, 3, 4, 5, 6}
segundo = {4, 5, 6, 7, 8, 9}

# Diferencia
print(primero - segundo)  # {1, 2, 3}
print(segundo - primero)  # {8, 9, 7}
```

El operador de diferencia simétrica ôbtiene los elementos que están en cualquiera de los conjuntos, pero no en ambos.

```
primero = {1, 2, 3, 4, 5, 6}
segundo = {4, 5, 6, 7, 8, 9}

# Diferencia simétrica
print(primero ^ segundo)  # {1, 2, 3, 7, 8, 9}
# Forma equivalente
print((primero | segundo) - (primero & segundo))  # {1, 2, 3, 7, 8, 9}
```

16.10. Estructuras de datos

Como se ha visto en capítulos anteriores, Python tiene soporte de las siguientes estructuras de datos: listas, diccionarios, tuplas y conjuntos.

¿Cúando utilizar diccionarios?

- Cuando se necesita utilizar asociaciones lógicas entre pares clave:valor.
- Cuando se necesita buscar datos rápidamente, en base a claves personalizadas.
- Cuando los datos son constantemente modificados.

¿Cúando utilizar listas?

- Cuando se tiene un grupo de datos que no necesita acceso aleatorio (deben estar ordenados).
- Cuando se necesita una recolección simple e iterable que es modificada frecuentemente.

¿Cúando utilizar conjuntos?

- Cuando se necesita que los elementos sean únicos.

¿Cúando utilizar tuplas?

- Cuando se necesita almacenar datos que no pueden ser cambiados.

En muchas ocasiones, una tupla es utilizada junto con un diccionario. Por ejemplo, una tupla puede representar una clave, porque es inmutable.

La siguiente tabla muestra las propiedades de cada estructura de datos.

	Lista	Diccionario	Tupla	Conjunto
Mutable	Sí	Sí	No	Sí
Indexado	Índices	Claves	Índices	No
Secuencial	Sí	No	Sí	No
Elementos únicos	No	No	No	Sí
Notación	[ ]	{ }	( )	{ }

Capítulo 17

Iterables

17.1. Cortes de lista

Los cortes de lista ofrecen una manera más avanzada de obtener valores de una lista. Los cortes de lista básicos involucran indexar una lista con dos enteros separados por dos puntos. Esto devuelve una lista nueva que contiene todos los valores de la lista vieja entre los índices.

```
cuadrados = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

print(cuadrados[2:6])  # [4, 9, 16, 25]
print(cuadrados[3:8])  # [9, 16, 25, 36, 49]
print(cuadrados[0:1])  # [0]
```

Como los argumentos de range, el primer índice provisto en un corte es incluido en el resultado, pero el segundo no.

Si el primer número en un corte es omitido, se toma el principio de la lista. Si el segundo número es omitido, se toma el final de la lista.

```
cuadrados = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

print(cuadrados[:5])  # [0, 1, 4, 9, 16]
print(cuadrados[5:])  # [25, 36, 49, 64, 81]
print(cuadrados[:])  # [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Los cortes de lista también pueden tener un tercer número, representando el incremento, para incluir valores alternativos en el corte.

```
cuadrados = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

print(cuadrados[::2])  # [0, 4, 16, 36, 64]
print(cuadrados[2:8:3])  # [4, 25]
```

Los valores negativos pueden ser utilizados en un corte de lista. Cuando los valores negativos son utilizados para el primer y el segundo valor del corte, estos cuentan desde el final de la lista.

```
cuadrados = [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

print(cuadrados[1:-1])  # [1, 4, 9, 16, 25, 36, 49, 64]
```

El tamaño de paso también puede ser negativo, lo que permite usarlos para invertir listas sin usar el método reverse().

```
lista = [1, 2, 3, 4]
lista = lista[::-1]
print(lista)  # [4, 3, 2, 1]
```

Es importante recordar que si el tamaño de paso es negativo, el primer número debe indicar un índice mayor que el segundo.

17.2. Cortes de tuplas

Los cortes también pueden ser realizados en tuplas. Su funcionamiento es el mismo que con listas.

```
tupla = (1, 2, 3, 4, 5, 6, 7, 8)

print(tupla[2:5])  # (3, 4, 5)
```

Esto es válido porque si bien las tuplas son inmutables, los cortes no realizan ningún cambio sobre los objetos con los que trabajan. Sólo consultan información de cierta manera.

17.3. Subcadenas

En Python pueden usarse cortes para obtener trozos de una cadena. Esto es lo que otros lenguajes de programación realizan usando el método substring(). Funciona de la misma forma que con listas y tuplas.

```
cadena = "Esta es una cadena"
print(cadena[2:10])  # ta es un
print(cadena[13:3:-1])  # ac anu se
```

17.4. Listas por compresión

Las listas por compresión son una forma útil de crear rápidamente listas cuyo contenido obedece una regla sencilla.

```python
# Lista por compresión
cubos = [i**3 for i in range(5)]

# Cubos de 0 a 4
print(cubos)
```

Las listas por compresión son inspiradas por la notación de constructores de conjuntos en las matemáticas.

Una lista por compresión también puede contener una sentencia if para aplicar una condición en los valores de la lista.

```python
cuadrados_pares = [i**2 for i in range(10) if i**2 % 2 == 0]

print(cuadrados_pares)  # [0, 4, 16, 36, 64]
```

Intentar crear una lista de rango demasiado extenso resultará en un MemoryError. Es posible que la operación tarde unos minutos en llegar a esta excepción.

```python
lista_extensa = [2 * i for i in range(10**100)]  # rango de 1 googol
```

Este problema es resuelto con generadores, los cuales se verán más tarde.

17.5. Formateo de cadenas

La forma más básica de combinar cadenas y objetos que no son cadenas es convirtiendo dichos objetos a cadenas y concatenando las cadenas.

El formateo de cadenas ofrece una manera más potente de incorporar objetos que no son cadenas a las cadenas. El formateo de cadenas utiliza el método format de una cadena para sustituir los argumentos de esta.

Una forma de hacerlo es usando índices.

```python
nums = [4, 5, 6]
mensaje = "Números: {0}, {1}, {2}".format(nums[0], nums[1], nums[2])
print(mensaje)
# Números: 4, 5, 6
```

Cada argumento de la función de formateo es colocada en la cadena de la posición correspondiente, que es determinada usando llaves .

Los índices entre llaves pueden repetirse cuantas veces se quiera.

```python
print("{0}{1}{0}".format("abra", "cad")) # abracadabra
```

Si se omite un índice, habrá algún argumento de format() que no se usará.

```
mensaje = "Números: {0}, {1}, {3}".format(4, 5, 6, 7)
print(mensaje)  # Números: 4, 5, 7
```

No es necesario escribir los índices dentro de las llaves , pero esto no permite que se repitan o que se entreguen como argumentos en otro orden.

```
mensaje = "Números: {}, {}, {}".format(10, 11, 12)
print(mensaje)  # Números: 10, 11, 12
```

El formateo de cadenas también puede hacerse con argumentos con nombre.

```
a = "{x}, {y}".format(x=5, y=12)
print(a)  # 5, 12
```

17.6. Funciones de cadenas

Python contiene muchas funciones integradas y métodos útiles que sirven para cumplir tareas comunes. Algunos métodos que se pueden usar con strings son:

- join(): Combina una lista de cadenas con otra cadena como separador.

```
print(" | ".join(["rojo", "azul", "amarillo"]))
# rojo | azul | amarillo
```

- replace(): Reemplaza una subcadena de una cadena por otra cadena.

```
print("Hola M".replace("M", "mundo"))
# Hola mundo
```

- startswith(): Determina si hay una subcadena al principio de una cadena.

```
frase = "Esta es una frase."
print(frase.startswith("Esta"))  # True
print(frase.startswith("Esta es"))  # True
```

- endswith(): Determina si hay una subcadena al final de una cadena.

```
frase = "Esta es una frase."
print(frase.endswith("una frase."))  # True
print(frase.endswith("."))  # True
```

- A diferencia de otros lenguajes de programación, Python no tiene el método contains(), para ver si una subcadena pertenece a un string.

  Usar el operador in tiene el mismo efecto que tendría dicha función.

- upper(): Cambia una cadena a mayúsculas.

```
frase = "Esta es una frase."
print(frase.upper())  # ESTA ES UNA FRASE.
```

- lower(): Cambia una cadena a minúsculas.

```
frase = "ESTA ES UNA FRASE."
print(frase.lower())  # esta es una frase.
```

- capitalize(): Cambia una cadena para que la primera letra sea mayúscula y el resto minúsculas.

```
frase = "ESTA ES UNA FRASE."
print(frase.capitalize())  # Esta es una frase.
```

- title(): Cambia una cadena para que todas las palabras empiecen con mayúscula.

```
frase = "ESTA ES UNA FRASE."
print(frase.title())  # Esta Es Una Frase.
```

17.7. Funciones numéricas

Para obtener la distancia entre un número y el 0 (su valor absulto), puede usarse la función abs().

```
print(abs(42))  # 42
print(abs(-42))  # 42
```

Para redondear un número a un determinado número de decimales, puede usarse la función round().

```python
from math import pi

print(round(pi, 0))  # 3.0
print(round(pi, 1))  # 3.1
print(round(pi, 2))  # 3.14
print(round(pi, 4))  # 3.1416
print(round(pi, 8))  # 3.14159265
```

17.8. Funciones all() y any()

All y any son utilizados con frecuencia como sentencias condicionales. Estos toman una lista como un argumento y devuelven True si todos o algunos (respectivamente) de sus argumentos son evaluados como True (o, de lo contrario, False).

```python
nums = [55, 44, 33, 22, 11]

if all([i > 5 for i in nums]):
    print('Todos los números son mayores que 5')

if any([i % 2 == 0 for i in nums]):
    print('Al menos un número es par')
```

17.9. Función enumerate()

La función enumerate() se usa para iterar a través de los valores e índices de una lista, simultáneamente.

```python
nums = [55, 44, 33, 22, 11]

for v in enumerate(nums):
  print(v)
# (0, 55)
# (1, 44)
# (2, 33)
# (3, 22)
# (4, 11)
```

También se puede especificar el índice desde el que comienza.

```python
frutas = ["manzana", "naranja", "pera", "limón", "durazno", "uva", "cereza"]

for f in enumerate(frutas, -3):
  print(f)
```

Capítulo 18

Programación funcional

18.1. Paradigma de programación funcional

La programación funcional es un estilo de programación que, como dice su nombre, gira en torno a funciones.

Una parte clave de la programación funcional son las funciones de orden superior. Similar al uso de funciones como objetos, las funciones de orden superior reciben otras funciones como argumentos, o las devuelven como resultado.

```
def repetir2veces(funcion, arg):
    return funcion(funcion(arg))

def sumar5(x):
    return x + 5

print(repetir2veces(sumar5, 10))  # 20
```

La función repetir2veces() recibe otra función como su argumento y llama 2 veces dentro de su cuerpo.

Entender los conceptos de programación funcional requiere un conocimiento básico del concepto algebraico llamado "composición de funciones".

18.2. Funciones puras

La programación funcional busca utilizar funciones puras. Las funciones puras no tienen efectos secundarios y devuelven un valor que depende únicamente de sus argumentos.

Así son las funciones en las matemáticas. Por ejemplo, la función cos(x) siempre devolverá un mismo resultado para el mismo valor de x.

Ejemplo de función pura:

```
def pura(x, y):
    temp = x + 2 * y
    return temp / (2 * x + y)
```

Ejemplo de función impura:

```
lista = []

def impura(arg):
    lista.append(arg)
```

Dicho de otra forma, una función pura cumple lo siguiente:

- Depende sólo de sus argumentos y de variables locales creadas dentro de ella.
- Siempre retorna el mismo resultado para los mismos argumentos.
- Se puede ejecutar en cualquier parte del programa sin causar efectos secundarios de ningún tipo.
- No altera ningún elemento fuera de ella.
- Puede usarse en otros programas y entrega los mismos resultados.

Utilizar funciones puras tiene sus ventajas y desventajas.

Las funciones puras son:

- Más fáciles de analizar y probar.
- Más eficientes. Una vez que la función haya sido evaluada para una entrada, el resultado puede ser almacenado y referenciado para la próxima vez que la función con esa entrada sea necesaria, reduciendo el número de veces que la función es llamada. Esto se denomina memorización.
- Más fáciles de ejecutar en paralelo.

Desventajas principales:

- Complican en su mayor parte la normalmente sencilla tarea de Entrada/Salida, ya que requiere de efectos secundarios inherentemente.
- En algunas situaciones, pueden ser más difíciles de escribir.

18.3. Lambdas

Crear una función normalmente (utilizando def) le asigna una variable automáticamente.

Esto es distinto a la creación de otros objetos, tales como cadenas y enteros, que pueden ser creados en el camino, sin la necesidad de asignarles una variable.

Lo mismo es posible con las funciones, dado que sean creadas utilizando la sintaxis lambda. Funciones creadas de esta forma son conocidas como anónimas. Este enfoque es más comúnmente utilizado cuando se pasa una función sencilla como argumento de otra función.

La sintaxis que se muestra a continuación consiste de la palabra reservada lambda seguida de una lista de argumentos, dos puntos, y una expresión a evaluar y devolver.

```
def func(f, arg):
    return f(arg)

y = func(lambda x: 2*x*x, 5)
print(y)  # 50
```

Las funciones lambda reciben su nombre del cálculo lambda, el cual es un modelo computacional inventado por Alonzo Church. Las funciones lambda no son tan potentes como las funciones con nombre. Sólo pueden hacer cosas que requieren de una sola expresión, normalmente equivalente a una sola línea de código.

```
# función con nombre
def polinomio(x):
    return x**2 + 5*x + 4
print(polinomio(-4))  # 0

# función lambda
print((lambda x: x**2 + 5*x + 4)(-4))  # 0
```

Las funciones lambda pueden ser asignadas a variables y ser utilizadas como funciones regulares.

```
doble = lambda x: x * 2
print(doble(7))  # 14
```

Sin embargo, rara vez existe una buena razón para hacer esto. Normalmente es mejor definir una función con def.

18.4. Función map()

La función map() es una función de orden superior muy útil que opera sobre listas (u objetos similares llamados iterables).

Esta función recibe una función y un iterable como argumentos y devuelve un nuevo iterable con la función aplicada a cada argumento.

```
def sumar5(x):
    return x + 5

nums = [11, 22, 33, 44, 55]
resultado = list(map(sumar5, nums))
print(resultado)  # [16, 27, 38, 49, 60]
```

El mismo resultado se puede obtener con mayor facilidad utilizando la sintaxis lambda.

```
nums = [11, 22, 33, 44, 55]
resultado = list(map(lambda x: x + 5, nums))
print(resultado)  # [16, 27, 38, 49, 60]
```

Los objetos del tipo mapa son iterables, por lo que se debe usar list() para convertir el resultado en una lista y poder verlo.

18.5. Función filter()

La función filter() es otra función de orden supero que se puede usar sobre iterables.

Esta función filtra un iterable eliminando elementos que no coincidan con el predicado (una función que devuelve un booleano).

```
nums = [11, 22, 33, 44, 55]
resultado = list(filter(lambda x: x % 2 == 0, nums))
print(resultado)  # [22, 44]
```

Al igual que map(), el resultado tiene que ser convertido explícitamente a una lista si se quiere imprimir.

18.6. Generadores

Los generadores son un tipo de iterable, como las listas o las tuplas.

A diferencia de las listas, no permiten indexar con índices arbitrarios, pero pueden aún ser iterados con bucles for.

Pueden ser creados utilizando funciones y la sentencia yield.

```
def cuenta_regresiva():
    i = 5
    while i > 0:
        yield i
        i -= 1

for i in cuenta_regresiva():
    print(i)
# muestra los números del 5 al 1
```

La sentencia yield es utilizada para definir un generador, reemplazando el retorno de una función para proveer un resultado a su llamador sin destruir las variables locales.

Debido al hecho que yield produce un elemento a la vez, los generadores no tienen las restricciones de memoria de las listas. De hecho, ¡pueden ser infinitos!

```
def infinitos_sietes():
    while True:
```

```
        yield 7

for i in infinitos_sietes():
    print(i)
```

En resumen, los generadores permiten declarar una función que se comporta como un iterador. En otras palabras, que puede utilizarse en un bucle for.

Un ejemplo de uso de generadores es para generar números primos.

```
from math import sqrt

def es_primo(num):
    for i in range(2, int(sqrt(num)) + 1):
        if num % i == 0:
            return False

    return True

def generar_primos():
    n = 2

    while True:
        if es_primo(n):
            yield n
        n += 1

for i in generar_primos():
    print(i)
```

Los generadores finitos pueden ser convertidos en listas al pasarlos como argumentos de la función list().

```
def numeros(x):
    for i in range(x):
        if i % 2 == 0:
            yield i

print(list(numeros(11)))  # [0, 2, 4, 6, 8, 10]
```

Utilizar generadores resulta en un mejor rendimiento, el cual es el resultado de una generación ociosa de valores (a medida que se vayan necesitando), lo cual se traduce en un uso reducido de memoria. Es más, no necesitamos esperar hasta que todos los elementos sean generados antes de empezar a utilizarlos.

18.7. Decoradores

Los decoradores ofrecen una forma de modificar funciones utilizando otras funciones. Esto es ideal cuando se necesita extender la funcionalidad de funciones que no se quieren modificar.

```
def decor(func):
    def envolver():
        print("=" * 12)
        func()
        print("=" * 12)
    return envolver

def imprimir_texto():
    print("¡Hola mundo!")

texto_decorado = decor(imprimir_texto)
texto_decorado()
# ============
# ¡Hola mundo!
# ============
```

En este caso, el uso la función envolver() se define dentro de decor() para permitir que decor() retorne una función, que es el objetivo principal de un decorador. La función texto_decorado() es una versión decorada de la función imprimir_texto().

De hecho, si se escribiera un decorador útil, podría reemplazarse imprimir_texto() por su versión decorada.

```
def decor(func):
    def envolver():
        print("=" * 12)
        func()
        print("=" * 12)
    return envolver

def imprimir_texto():
    print("¡Hola mundo!")

imprimir_texto()
# ¡Hola mundo!

imprimir_texto = decor(imprimir_texto)
imprimir_texto()
# ============
# ¡Hola mundo!
# ============

imprimir_texto = decor(imprimir_texto)
imprimir_texto()
# ============
# ============
# ¡Hola mundo!
# ============
# ============
```

En el ejemplo anterior, se decora la función print_text() reemplazando la variable que contiene la función por una versión envuelta.

```
def imprimir_texto():
    print("¡Hola mundo!")

imprimir_texto = decor(imprimir_texto)
```

Este patrón puede utilizarse en cualquier momento, para envolver cualquier función.

Python ofrece apoyo para envolver una función en un decorador anteponiendo la definición de la función con el nombre de un decorador y el símbolo , lo cual tendrá el mismo resultado que el código de arriba.

```
@decor  # nombre del decorador
def imprimir_texto():
    print("¡Hola mundo!")
```

Una sola función puede tener varios decoradores y cada decorador puede repetirse más de una vez.

```
def decor1(func):
    def envolver():
        print("=" * 12)
        func()
        print("=" * 12)
    return envolver

def decor2(func):
    def envolver():
        print("*" * 12)
        func()
        print("*" * 12)
    return envolver

@decor2
@decor1
@decor2
def imprimir_texto():
    print("¡Hola mundo!")

imprimir_texto()
# ************
# ============
# ************
# ¡Hola mundo!
# ************
# ============
# ************
```

Los decoradores pueden ser usados para muchos otros propósitos además de “decorar”.

```
def entrada(func):
    def envolver():
```

```
        x = int(input(': '))
        y = int(input(': '))
    func(x, y)
    return envolver

@entrada
def sumar(x, y):
    print(x + y)

@entrada
def restar(x, y):
    print(x - y)

@entrada
def multiplicar(x, y):
    print(x * y)

# Ya no necesitan argumentos para llamarse
sumar()
restar()
multiplicar()
```

```
import time

def tiempo_transcurrido(f):
    def envolver(*n):  # *n significa cualquier número de argumentos
        t1 = time.time()
        f(*n)
        t2 = time.time()
        tiempo = (t2 - t1) * 1000
        print("Tiempo transcurrido: {} ms".format(tiempo))
    return envolver

@tiempo_transcurrido
def suma_grande():
    numeros = [i for i in range(0, 1000000)]
    print("Suma: {}".format(sum(numeros)))

suma_grande()
```

18.8. Recursión

La recursión es un concepto muy importante en la programación funcional.

Lo fundamental de la recursión es la autorreferencia, funciones que se llaman a sí mismas. Se utiliza para resolver problemas que pueden ser divididos en subproblemas más sencillos del mismo tipo.

Un ejemplo clásico de una función que es implementada recursiva es la función factorial.

```
def factorial(n):
    if n < 0:
```

```
        raise ValueError
    if n == 1 or n == 0:
        return 1
    else:
        return n * factorial(n-1)

print(factorial(5))  # 120
print(factorial(0))  # 1
print(factorial(-10))  # ValueError
```

El caso base n == 0 actúa como condición de salida de la recursión, porque no involucra más llamadas a la función.

Las funciones recursivas pueden ser infinitas, al igual que los bucles while. Estas ocurren cuando se olvida implementar algún caso base.

Abajo se muestra una versión incorrecta de la función factorial. No tiene caso base, así que se ejecuta hasta que al interpretador se le acabe la memoria o se cuelgue.

```
def factorial(n):
    return n * factorial(n-1)

print(factorial(5))
```

La recursión también puede ser indirecta. Una función puede llamar a una segunda, que a su vez llama a la primera, que llama a la segunda, y así sucesivamente. Esto puede ocurrir con cualquier cantidad de funciones.

```
def es_par(n):
    if n == 0:
        return True
    else:
        return es_impar(n-1)

def es_impar(n):
    return not es_par(n)

print(es_impar(17))  # True
print(es_par(23))  # False
```

Otro ejemplo clásico es la serie de Fibonacci.

```
def fibonacci(n):
    if n < 0:
        raise ValueError
    elif n == 0 or n == 1:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)

```

```
for i in range(0, 10):
    print(fibonacci(i))
```

18.9. Iteración vs. Recursión

Capítulo 19

El módulo itertools

19.1. El módulo itertools

El módulo itertools es una biblioteca estándar que contiene varias funciones que son útiles en la programación funcional.

```
import itertools
```

19.2. Iteradores infinitos

Uno de los tipos de función que produce son iteradores infinitos.

La función count() cuenta infinitamente a partir de un valor.

```
from itertools import count

for i in count(3):  # Empieza desde 3
    print(i)
    if i >= 10:  # Termina en 10
        break
```

Se le puede entregar un segundo parámetro a count(), el cual representa el tamaño de paso. Si no se le entrega el segundo parámetro, el tamaño de paso por defecto es 1.

```
from itertools import count

for i in count(3, 2):  # Empieza desde 3, contando de 2 en 2
    print(i)
    if i >= 20:  # Termina en 20
        break
```

Si el segundo parámetro es negativo, cuenta hacia atrás.

```python
from itertools import count

for i in count(100, -5):  # Empieza desde 3, contando de -5 en -5
    print(i)
    if i >= 20:  # Termina en 20
        break
```

La función cycle() itera infinitamente a través de un iterable (como una lista o cadena).

```python
from itertools import cycle

for i in cycle("hola"):
    print(i)
# h
# o
# l
# a
# h
# o
# l
# a
# ...
```

```python
from itertools import cycle

for i in cycle([11, "hola", 33.5, False]):
    print(i)
# 11
# hola
# 33.5
# False
# 11
# hola
# 33.5
# False
# ...
```

La función repeat() repite un objeto, ya sea infinitamente o un número específico de veces.

```python
from itertools import repeat

for i in repeat("hola", 3):
    print(i)
# hola
# hola
# hola
```

Puede usarse para crear listas que tengan un objeto repetido.

```
from itertools import repeat

holas = list(repeat("hola", 3))
print(holas)  # ['hola', 'hola', 'hola']
```

Para repetir infinitamente, se le debe entregar sólo el objeto como parámetro.

```
from itertools import repeat

for i in repeat("hola"):
    print(i)
# hola
# hola
# hola
# ...
```

19.3. Operaciones sobre iterables

Hay muchas funciones en itertools que operan sobre iterables, de una forma similar a map() o filter().

La función accumulate() devuelve un total actualizado de los valores dentro de un iterable.

```
from itertools import accumulate

nums = list(accumulate(range(8)))

print(list(range(8)))  # [0, 1, 2, 3, 4, 5, 6, 7]
print(nums)  # [0, 1, 3, 6, 10, 15, 21, 28]
```

La función takewhile() toma elementos de un iterable mientras una función predicado permanece verdadera.

```
from itertools import takewhile

nums = range(10)

print(list(nums))  # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(list(takewhile(lambda x: x <= 5, nums)))  # [0, 1, 2, 3, 4, 5]
```

La diferencia entre takewhile() y filter() es que takewhile() deja de tomar elementos cuando llega al primer elemento que no cumple la condición de la función.

```python
from itertools import takewhile

nums = [1, 2, 3, 4, 5, 4, 3, 2, 1]
print(list(takewhile(lambda x: x <= 3, nums)))  # [1, 2, 3]
print(list(filter(lambda x: x <= 3, nums)))  # [1, 2, 3, 3, 2, 1]
```

La función chain() combina varios iterables en uno solo más largo.

```python
from itertools import chain

lista1 = [1, 2, 3]
lista2 = [4, 5, 6]

print(list(chain(lista1, lista2)))  # [1, 2, 3, 4, 5, 6]
```

19.4. Funciones de combinatoria

También hay numerosas funciones combinatorias en itertools, tales como product() y permutations(). Estas son usadas cuando se quiere cumplir tareas con todas las combinaciones posibles de algunos elementos.

La función product() retorna el producto entre 2 iterables.

```python
from itertools import product

letras1 = ("A", "B")
letras2 = ("C", "D")

print(list(product(letras1, letras2)))
# [('A', 'C'), ('A', 'D'), ('B', 'C'), ('B', 'D')]
```

La función permutations() retorna las todas permutaciones posibles entre los elementos de un iterable.

```python
from itertools import permutations

letras  = ("A", "B")

print(list(permutations(letras)))
# [('A', 'B'), ('B', 'A')]
```

```python
from itertools import permutations

letras = ("A", "B", "C")

```

```
print(list(permutations(letras)))
# [('A', 'B', 'C'), ('A', 'C', 'B'), ('B', 'A', 'C'), ('B', 'C', 'A'), ('C',
↪  'A', 'B'), ('C', 'B', 'A')]
```

Se le puede entregar un segundo argumento, el cual representa el número de elementos que cada permutación debe tener.

```
from itertools import permutations

letras  = ("A", "B")

print(list(permutations(letras, 1)))
# [('A',), ('B',)]
print(list(permutations(letras, 2)))
# [('A', 'B'), ('B', 'A')]
print(list(permutations(letras, 3)))
# [] (porque es imposible, ya que sólo tiene 2 elementos)
```

El método combinations() funciona de forma similar a permutations(), pero muestra las combinaciones en vez de permutaciones.

```
from itertools import combinations

letras  = ("A", "B", "C")

print(list(combinations(letras, 1)))
# [('A',), ('B',), ('C',)]
print(list(combinations(letras, 2)))
# [('A', 'B'), ('A', 'C'), ('B', 'C')]
print(list(combinations(letras, 3)))
# [('A', 'B', 'C')]
print(list(combinations(letras, 4)))
# [] (porque es imposible, ya que sólo tiene 3 elementos)
```

La diferencia es que en las combinaciones no importa el orden y se unen todos los resultados que tienen los mismos elementos pero en orden distinto.

Capítulo 20

Programación orientada a objetos

20.1. Programación orientada a objetos

Anteriormente se vieron 2 paradigmas de programación: imperativa (utilizando declaraciones, bucles y funciones como subrutinas) y funcional (utilizando funciones puras, funciones de orden superior y recursión).

Otro paradigma muy popular es la programación orientada a objetos (POO). Los objetos son creados utilizando clases, las cuales son en realidad el eje central de la POO.

20.2. Clases

La clase describe lo que el objeto será, pero es independiente del objeto mismo. En otras palabras, una clase puede ser descrita como los planos, la descripción o definición de un objeto. Una misma clase puede ser utilizada como plano para crear varios objetos diferentes.

Las clases son creadas utilizando la palabra clave class y un bloque indentado que contiene los métodos de una clase (los cuales son funciones).

```
1 class Gato:
2     def __init__(self, color):
3         self.color = color
4
5 felix = Gato("negro")
6 tigre = Gato("café")
```

El código define una clase llamada Gato, la cual tiene el atributo color. Luego, la clase es utilizada para crear 2 objetos independientes de esa clase.

20.3. Método __init__

El método __init__ es el más importante de una clase. Es llamada cuando una instancia (objeto) de una clase es creada, utilizando el nombre de la clase como función.

Todos los métodos deben tener self como su primer parámetro. Aunque no sea pasado explícitamente, Python agrega el argumento self automáticamente. No se necesita entregar cuando se llaman los métodos.

Dentro de la definición de un método, self se refiere a la instancia que está llamando al método.

Las instancias de una clase tienen atributos, los cuales son datos asociados a ellas. En este ejemplo, las instancias de Gato tienen los atributos color y edad. Los atributos pueden ser accedidos al poner un punto seguido del nombre del atributo luego del nombre de una instancia.

```
class Gato:
    # Constructor
    def __init__(self, color, edad):
        self.color = color
        self.edad = edad

felix = Gato("café", 7)  # Instancia (objeto) de la clase Gato

print(felix.color)  # café
print(felix.edad)  # 7
```

En un método __init__, self.atributo puede ser usado para fijar un valor inicial a los atributos de una instancia.

En el método mostrado anteriormente, el método __init__ recibe 2 argumentos y los asigna a los atributos del objeto. El método __init__ es llamado el constructor de la clase.

Si una clase no tiene atributos que se quieran inicializar con cada instancia, se puede omitir el método __init__.

```
class A:
    valor = 1

a = A()
print(a.valor)  # 1
```

20.4. Atributos

20.5. Métodos

Las clases pueden tener otros métodos definidos para agregarles funcionalidad. Todos los métodos deben tener self como su primer parámetro. Estos métodos son accedidos utilizando la misma sintaxis de punto que los atributos.

```
class Perro:
    def __init__(self, nombre, color):
        self.nombre = nombre
        self.color = color

    def ladrar(self):
        print("¡Guau!")

fido = Perro("Fido", "blanco")

print(fido.nombre)  # Fido
```

```
print(fido.color)  # blanco
fido.ladrar()  # ¡Guau!
```

20.6. Atributos de clase

Las clases pueden tener atributos de clase también, creados al asignar variables dentro del cuerpo de una clase. Estos pueden ser accedidos desde instancias de una clase o desde la clase misma.

```
class Perro:
    # atributo de clase
    patas = 4

    # constructor
    def __init__(self, nombre, color):
        self.nombre = nombre
        self.color = color

fido = Perro("Fido", "blanco")

print(fido.patas)  # 4
print(Perro.patas)  # 4
```

Los atributos de clase son compartidos por todas las instancias de una clase. Realizar algún cambio a un atributo de la clase también hará ese cambio en las instancias de esa clase.

```
fido = Perro("Fido", "blanco")

print(fido.patas)  # 4
print(Perro.patas)  # 4

Perro.patas = 5

print(fido.patas)  # 5
print(Perro.patas)  # 5
```

20.7. Excepciones de clases

Tratar de acceder a un atributo de una instancia que no está definida generará un AttributeError. Esto también aplica cuando se llama un método no definido.

```
class Rectangulo:
    def __init__(self, ancho, altura):
        self.ancho = ancho
        self.altura = altura

rect = Rectangulo(5, 6)
```

```
print(Rectangulo.color)  # AttributeError
Rectangulo.pintar("azul")  # AttributeError
```

20.8. Herencia

La herencia brinda una manera de compartir funcionalidades entre clases.

Por ejemplo, las clases Perro, Gato, Conejo, etc. tienen algo en común. Aunque presenten algunas diferencias, también tienen muchas características en común. Este parecido puede ser expresado haciendo que todos hereden de una superclase Animal, que contiene las funcionalidades compartidas.

Para heredar de una clase desde otra, se coloca el nombre de la superclase entre paréntesis luego del nombre de la clase.

```
class Animal:
    def __init__(self, nombre, color):
        self.nombre = nombre
        self.color = color

class Gato(Animal):
    def maullar(self):
        print("¡Miau!")

class Perro(Animal):
    def ladrar(self):
        print("¡Guau!")

firulais = Perro("Firulais", "café")

print(firulais.nombre)  # Firulais
print(firulais.color)  # café
firulais.ladrar()  # ¡Guau!

misifu = Gato("Misifu", "blanco")

print(misifu.nombre)  # Misifu
print(misifu.color)  # Blanco
misifu.maullar()  # ¡Miau!
```

Una clase que hereda de otra clase se llama subclase. Una clase de la cual se hereda se llama superclase.

Si una clase hereda de otra con los mismos atributos o métodos, los sobreescribe.

```
class Lobo:
    def __init__(self, nombre, color):
        self.nombre = nombre
        self.color = color

    def ladrar(self):
        print("Grrr...")
```

```
8
9  class Perro(Lobo):
10     def ladrar(self):
11         print("Guau")
12
13 husky = Perro("Max", "gris")
14 husky.bark()  # Guau
```

La herencia también puede ser indirecta. Una clase hereda de otra, y esa clase puede a su vez heredar de una tercera clase.

```
1  class A:
2      def metodo1(self):
3          print("Método de A")
4
5  class B(A):
6      def metodo2(self):
7          print("Método de B")
8
9  class C(B):
10     def metodo3(self):
11         print("Método de C")
12
13 c = C()
14
15 c.metodo1()  # Método de A
16 c.metodo2()  # Método de B
17 c.metodo3()  # Método de C
```

Sin embargo, no es posible la herencia circular.

20.9. Función super()

La función super() es una útil función relacionada con la herencia que hace referencia a la clase padre. Puede ser utilizada para encontrar un método con un determinado nombre en la superclase del objeto.

```
1  class A:
2      valor = 1
3
4  a = A()
5  print(a.valor) # 1
6
7  class A:
8      def metodo(self):
9          print("A")
10
11 class B(A):
12     def metodo(self):
13         print("B")
14
```

```
    def metodo_super(self):
        super().metodo()

b = B()

b.metodo()
# B
b.metodo_super()
# A
```

También se puede usar para llamar al constructor de la superclase.

```
class Humano:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

    def presentar(self):
        print("Hola, soy {}".format(self.nombre))
        print("Tengo {} años".format(self.edad))

class Trabajador(Humano):
    def __init__(self, nombre, edad, trabajo):
        super().__init__(nombre, edad)
        self.trabajo = trabajo

    def presentar(self):
        super().presentar()
        print("Soy {}".format(self.trabajo))

t = Trabajador("John", 30, "profesor")
t.presentar()
# Hola, soy John
# Tengo 30 años
# Soy profesor
```

20.10. Métodos mágicos

Los métodos mágicos son métodos especiales que tienen doble guión bajo al principio y al final de sus nombres. Son también conocidos en inglés como dunders (de double underscores).

El constructor __init__ es un método mágico, pero existen muchos más. Son utilizados para crear funcionalidades que no pueden ser representadas en un método regular.

20.11. Sobrecarga de operadores aritméticos

Un uso común de métodos mágicos es la sobrecarga de operadores. Esto significa definir operadores para clases personalizadas que permiten que operadores tales como + y * sean utilizados en ellas.

El método mágico __add__ permite sobrecargar el operador +, lo cual permite darle un comportamiento personalizado.

```python
class Vector2D:
  def __init__(self, x, y):
      self.x = x
      self.y = y

  def __add__(self, otro):
      return Vector2D(self.x + otro.x, self.y + otro.y)

v1 = Vector2D(5, 7)
v2 = Vector2D(3, 9)
resultado = v1 + v2

print(resultado.x)  # 8
print(resultado.y)  # 16
```

El método __add__ suma los atributos correspondientes de los objetos y devuelve un nuevo objeto que contiene el resultado. Una vez definido, se pueden sumar dos objetos de una clase entre sí.

Otra forma de verlo es que el intérprete de Python siempre interpretará el operador + como el método __add__ de su clase.

Los métodos mágicos para operadores comunes son:

```python
__add__  # operador +
__sub__  # operador -
__mul__  # operador *
__truediv__  # operador /
__floordiv__  # operador //
__mod__  # operador %
__pow__  # operador **
__and__  # operador &
__xor__  # operador ^
__or__  # operador |
```

Y después el intérprete de Python interpreta los operadores así:

```python
# Si x e y son del mismo tipo y este tipo tiene implementado los métodos
↪ mágicos
x + y == x.__add__(y)
x - y == x.__sub__(y)
x * y == x.__mul__(y)
x / y == x.__truediv__(y)
x // y == x.__floordiv__(y)
x % y == x.__mod__(y)
x ** y == x.__pow__(y)
x & y == x.__and__(y)
x ^ y == x.__xor__(y)
x | y == x.__or__(y)
```

A continuación, se muestra un ejemplo de implementación del método __truediv__.

```python
class CadenaEspecial:
    def __init__(self, contenido):
        self.contenido = contenido

    def __truediv__(self, otro):
        linea = "=" * len(otro.contenido)
        return "\n".join([self.contenido, linea, otro.contenido])

palabra1 = CadenaEspecial("hola")
palabra2 = CadenaEspecial("mundo")

print(palabra1 / palabra2)
# hola
# =====
# mundo
```

La sobrecarga de operadores no necesita que los métodos mágicos nuevos cumplan su misma función original. Un caso particular ocurre cuando "x" e "y" son de tipos distintos y "x" no tiene definido un método mágico, por ejemplo __add__. En este caso, Python intentará calcular la operación al revés, la cual en este caso será y + x. Para realizar esta operación llamará al método reverso __radd__ de "y". Hay métodos r equivalentes para todos los operadores mencionados anteriormente.

```python
# Si x e y son de tipo distinto y x no ha implementado los métodos mágicos
x + y == y.__radd__(x)
x - y == y.__rsub__(x)
x * y == y.__rmul__(x)
x / y == y.__rtruediv__(x)
x // y == y.__rfloordiv__(x)
x % y == y.__rmod__(x)
x ** y == y.__rpow__(x)
x & y == y.__rand__(x)
x ^ y == y.__rxor__(x)
x | y == y.__ror__(x)
```

Y un ejemplo de implementación de métodos reversos:

```python
class Perro:
    def __init__(self, nombre):
        self.nombre = nombre

class Gato:
    def __init__(self,nombre):
        self.nombre = nombre

    def __radd__(self, otro):
        # Perro + Gato = string
        return "\t".join([self.nombre, otro.nombre])

perro = Perro("Joe")
gato = Gato("Ollie")

print(perro + gato)
```

```
# Ollie          Joe
```

20.12. Sobrecarga de operadores de comparación

Python también ofrece métodos mágicos para comparaciones.

```
__lt__  # comparador <, less than
__le__  # comparador <=, less equal
__eq__  # comparador ==, equal
__ne__  # comparador !=, not equal
__gt__  # comparador >, greater than
__ge__  # comparador >=, greater equal
```

Si __ne__ no está implementado, devuelve el opuesto de __eq__. No hay ninguna otra relación entre los otros operadores.

```
class CadenaEspecial:
    def __init__(self, contenido):
        self.contenido = contenido

    def __gt__(self, otro):
        for i in range(len(otro.contenido) + 1):
            resultado = otro.contenido[:i] + ">" + self.contenido
            resultado += ">" + otro.contenido[i:]
            print(resultado)

palabra1 = CadenaEspecial("hola")
palabra2 = CadenaEspecial("mundo")

palabra1 > palabra2
# >hola>mundo
# m>hola>undo
# mu>hola>ndo
# mun>hola>do
# mund>hola>o
# mundo>hola>
```

20.13. Métodos mágicos de contenedores

Hay varios métodos mágicos para hacer que las clases actúen como contenedores.

```
__len__  # método len()
__getitem__  # indexar
__setitem__  # asignar valores indexados
__delitem__  # borrar valores indexados
```

```
__iter__  # iteración sobre objetos (por ejemplo en bucles for)
__contains__  # operador in
```

A continuación, se muestra un ejemplo rebuscado pero creativo, el cual consiste en crear una clase de lista poco confiable o imprecisa.

```
import random

class ListaImprecisa:
    def __init__(self, contenido):
        self.contenido = contenido

    def __getitem__(self, indice):
        return self.contenido[indice + random.randint(-1, 1)]

    def __len__(self):
        return random.randint(0, len(self.contenido) * 2)

lista_imprecisa = ListaImprecisa(["A", "B", "C", "D", "E"])

# Los resultados son aleatorios, haciendo que la lista sea imprecisa
print(len(lista_imprecisa))
print(len(lista_imprecisa))
print(len(lista_imprecisa))
print(lista_imprecisa[2])
print(lista_imprecisa[2])
print(lista_imprecisa[2])
```

20.14. Ciclo de vida de un objeto

El ciclo de vida de un objeto está conformado por su creación, manipulación y destrucción.

La primera etapa del ciclo de vida de un objeto es la definición de la clase a la cual pertenece.

La siguiente etapa es la instanciación de un objeto, cuando el método __init__ es llamado. La memoria es asignada para almacenar la instancia. Justo antes de que esto ocurra, el método __new__ de la clase es llamado, para asignar la memoria necesaria. Este es normalmente redefinido sólo en casos especiales.

Luego de que ocurra lo anterior el objeto estará listo para ser utilizado.

Otro código puede interactuar con el objeto, llamando sus métodos o accediendo a sus atributos. Eventualmente, terminará de ser utilizado y podrá ser destruido.

Cuando un objeto es destruido, la memoria asignada se libera y puede ser utilizada para otros propósitos.

La destrucción de un objeto ocurre cuando su contador de referencias llega a cero. La cuenta de referencias es el número de variables y otros elementos que se refieren al objeto.

Si nada se está refiriendo al objeto (tiene una cuenta de referencias de 0) nada puede interactuar con este, así que puede ser eliminado con seguridad. En algunas situaciones, dos (o más) objetos pueden solo referirse entre ellos, y por lo tanto pueden ser eliminados también.

La sentencia del reduce la cuenta de referencias de un objeto por 1, y a menudo conlleva a su eliminación. El método mágico de la sentencia del es __del__.

El proceso de eliminación de objetos cuando ya no son necesarios se denomina recolección de basura (garbage collection).

En resumen, el contador de referencias de un objeto se incrementa cuando se le es asignado un nuevo nombre o es colocado en un contenedor (una lista, tupla o diccionario). La cuenta de referencias de un objeto se disminuye cuando es eliminado con del, su referencia es reasignada, o su referencia sale fuera del alcance. Cuando la cuenta de referencias de un objeto llega a 0, Python lo elimina automáticamente.

```
a = 42  # Creación del objeto <42>
b = a  # Aumenta el contador de referencias de <42>
c = [a]  # Aumenta el contador de referencias de <42>

del a  # Reduce el contador de referencias de <42>
b = 100  # Reduce el contador de referencias de <42>
c[0] = -1  # Reduce el contador de referencias de <42>

# <42> ya puede ser eliminado
```

Lenguajes de bajo nivel como C no tienen esta clase de manejo de memoria automático.

20.15. Ocultamiento de información

Un componente clave de la programación orientada a objetos es el encapsulamiento, que involucra empaquetar las variables y funciones relacionadas en un único objeto fácil de usar, una instancia de una clase.

Un concepto asociado es el de ocultamiento de información, el cual dicta que los detalles de implementación de una clase deben estar ocultos y que sean presentados a aquellos que quieran utilizar la clase en una interfaz estándar limpia. En otros lenguajes de programación, esto se logra normalmente utilizando métodos y atributos privados, los cuales bloquean el acceso externo a ciertos métodos y atributos en una clase.

La filosofía de Python es ligeramente diferente. A menudo se dice "todos somos adultos consistentes aquí", que significa que no deberías poner restricciones arbitrarias al acceso de las partes de una clase. Por ende, no hay formas de imponer que un método o atributo sea estrictamente privado.

Sin embargo, hay maneras de desalentar a la gente de acceder a las partes de una clase, tales como denotar que es un detalle de implementación y debe ser utilizado a su cuenta y riesgo.

Los métodos y atributos débilmente privados tienen un único guión bajo al principio. Esto señala que son privados, y no deberían ser utilizados por código externo.

Sin embargo, es en su mayor parte sólo una convención, y no impide que el código externo los acceda. Su único efecto verdadero es que `from nombre_de_modulo import *` no importará a las variables que empiecen por un único guión bajo.

20.16. Métodos de clase

20.17. Métodos estáticos

20.18. Propiedades

Capítulo 21

Expresiones regulares

21.1. Expresiones regulares

Capítulo 22

Empaquetamiento

Capítulo 23

Interfaz gráfica

Capítulo 24

Algoritmos de ordenamiento

24.1. Bubble sort

Capítulo 25

Algoritmos de búsqueda

Capítulo 26

Algoritmos de matrices

Capítulo 27

Implementación de estructuras de datos

Capítulo 28

La librería NumPy