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. 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.

El Zen de Python se muestra en pantalla la primera vez que se ejecute la siguiente linea.

```
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!
```

1.3. Hola mundo

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

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

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

```
1 print('Hola')
2 print('Mundo')
```

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)  # 2
print(5 - 2)  # 3
print(2 * 2)  # 4
print(4 / 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
4. Adición y Sustracción

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
```

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)  # 0.25
print(3 / 4)  # 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)  # 30000000000000004
```

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)  # 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.

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)  # 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)  # 3
```

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

```
print(7 % 2)  # 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)  # 0, 10 es múltiplo de 2
print(15 % 3)  # 0, 15 es múltiplo de 3
print(16 % 7)  # 2, 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")
```

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

```
# Strings no válidos
print('texto")
print("texto')
```

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('\\')  # \
```

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"')
```

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')
# 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('Hola\tmundo')
# Hola    mundo
```

  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, eliminando todos los caracteres de esa línea. Representado por \r.

```
print('12345\r67')
# 67
```

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

```
print('123\b45')
# 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')  # ñ
print('\u00F1')  # 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""")
# 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

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")  # Hola mundo
print("1" + "1")  # 11
```

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

```
# print(1 + "1")  # TypeError
```

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

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.

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.

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.

Ambos argumentos se pueden combinar.

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 =.

```
1 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.

```
1 123 = 44  # SyntaxError
2 123variable = 'hola'  # SyntaxError
3 hola mundo = 'hola mundo'  # SyntaxError
```

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

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)  # 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)
```

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(int(x))  # 7
```

- float(): Convierte a un float.

```
num = 42
print(float(num))  # 42.0
```

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

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

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 **.

```
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.

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

Capítulo 5

Declaraciones if

5.1. Booleanos

5.2. Operadores de comparación

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

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.

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

- 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

7.6. Rangos

7.7. Bucle for en rangos

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.

8.2. Definición de funciones

8.3. Llamado de funciones

8.4. Devolución de valores en una función

8.5. Docstring

8.6. Funciones como objetos

8.7. Sobrecarga de funciones

8.8. Anotaciones de tipos

Capítulo 9

Módulos y la biblioteca estándar

9.1. Módulos

9.2. Alias

9.3. 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:

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.

9.4. 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.

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.

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

Manejo de excepciones

Capítulo 13

Pruebas unitarias

Capítulo 14

Manejo de archivos

Capítulo 15

Módulos time y datetime

Capítulo 16

Iterables

16.15. Función enumerate()

Capítulo 17

Programación funcional

Capítulo 18

Conjuntos y estructuras de datos

Capítulo 19

El módulo itertools

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).

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.

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__.

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.

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.

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.

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.

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.

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.

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

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.

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

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

20.12. Sobrecarga de operadores de comparación

Python también ofrece métodos mágicos para comparaciones.

Si __ne__ no está implementado, devuelve el opuesto de __eq__. No hay ninguna otra relación entre los otros operadores.

20.13. Métodos mágicos de contenedores

Hay varios métodos mágicos para hacer que las clases actúen como contenedores.

A continuación, se muestra un ejemplo rebuscado pero creativo, el cual consiste en crear una clase de lista poco confiable o imprecisa.

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.

Lenguajes de bajo nivel como C no tienen esta clase de manejo de memoria automático.

20.15. Encapsulamiento

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.

Capítulo 21

Expresiones regulares

Capítulo 22

Empaquetamiento

Capítulo 23

Interfaz gráfica

Capítulo 24

Algoritmos de ordenamiento

Capítulo 25

Algoritmos de búsqueda

Capítulo 26

Algoritmos de matrices

Capítulo 27

Implementación de estructuras de datos

Capítulo 28

La librería NumPy