



Resumen Python 3

Víctor Mardones Bravo

Febrero de 2021

Índice general

1. Introducción a Python	1
1.1. ¿Qué es Python?	1
1.2. El Zen de Python	1
1.3. Hola mundo	2
2. Conceptos básicos	3
2.1. Comentarios	3
2.2. Operaciones aritméticas	3
2.3. La regla PEMDAS	3
2.4. Paréntesis	4
2.5. Floats	4
2.6. Exponenciación	4
2.7. Cociente y resto	5
3. Cadenas de texto	6
3.1. Strings o cadenas de caracteres	6
3.2. Cadena vacía	6
3.3. Caracteres especiales	6
3.4. Secuencias de escape	7
3.5. Caracteres Unicode	8
3.6. Strings multilínea	8
3.7. Comentarios multilínea	9
3.8. Concatenación de strings	9
3.9. Multiplicación de strings	9
3.10. Opciones del método print()	9
4. Variables	11
4.1. Asignación de variables	11
4.2. Nombre de variables válidos	11
4.3. Palabras clave	11
4.4. Operaciones con variables	12
4.5. Entrada	12
4.6. Conversión de tipos de datos	13
4.7. Operadores de asignación	13
5. Declaraciones if	14
5.1. Booleanos	14
5.2. Operadores de comparación	14
5.3. Declaración if	14
5.4. Declaración if-else	15
5.5. Declaración elif	15
5.6. Operadores lógicos	16
5.7. Precedencia de operadores lógicos	16
6. Listas	17
6.1. Creación de listas	17
6.2. Indexación de listas	17
6.3. IndexError en listas	18

6.4.	Lista vacía	18
6.5.	Anidación de listas	18
6.6.	Comas sobrantes	18
6.7.	Operaciones con listas	19
6.8.	Operadores in y not in	19
6.9.	Funciones de listas	19
6.10.	Copiar listas	22
6.11.	Strings como listas	22
6.12.	Indexación de strings	23
6.13.	Conversión de strings a listas	23
7.	Bucles	24
7.1.	Bucles while	24
7.2.	Bucles infinitos	24
7.3.	Declaración break	24
7.4.	Declaración continue	25
7.5.	Bucle for con listas	25
7.6.	Rangos	25
7.7.	Bucle for en rangos	26
8.	Funciones	27
8.1.	¿Qué es una función?	27
8.2.	Definición de funciones	27
8.3.	Llamado de funciones	27
8.4.	Devolución de valores en una función	27
8.5.	Docstring	28
8.6.	Funciones como objetos	28
8.7.	Sobrecarga de funciones	28
8.8.	Anotaciones de tipos	28
9.	Módulos y la biblioteca estándar	29
9.1.	Módulos	29
9.2.	Error de importación	29
9.3.	Alias	29
9.4.	La biblioteca estándar	29
9.5.	Módulos externos y pip	30
10.	El módulo math	31
11.	El módulo random	32
12.	Manejo de excepciones	33
12.1.	Excepciones	33
12.2.	Declaración try-except	33
12.3.	Declaración finally	34
12.4.	Levantar excepciones	34
12.5.	Aserciones	34
13.	Pruebas unitarias	35
14.	Manejo de archivos	36
14.1.	Abrir archivos	36
14.2.	Modos de apertura	36
14.3.	Cierre de archivos	36
14.4.	Lectura de archivos	36
14.5.	Escritura de archivos	37
14.6.	Declaración with	37
15.	Módulos time y datetime	38

16.Iterables	39
16.1. Objeto None	39
17.Diccionarios y tuplas	40
17.1. Diccionarios	40
17.2. Indexación de diccionarios	40
17.3. Uso de in y not en diccionarios	40
17.4. Función get()	40
17.5. Función keys()	41
17.6. Tuplas	41
18.Manejo de iterables	42
18.1. Cortes de lista	42
18.2. Cortes de tuplas	42
18.3. Subcadenas	42
18.4. Listas por compresión	43
18.5. Formateo de cadenas	43
18.6. Funciones de cadenas	43
18.7. Funciones numéricas	43
18.8. Funciones all() y any()	43
18.9. Función enumerate()	43
19.Programación funcional	44
19.1. Paradigma de programación funcional	44
19.2. Funciones puras	44
19.3. Lambdas	44
19.4. Función map()	45
19.5. Función filter()	45
19.6. Generadores	45
19.7. Decoradores	46
19.8. Recursión	46
19.9. Iteración vs. Recursión	46
20.Conjuntos y estructuras de datos	47
20.1. Conjuntos	47
20.2. Operaciones con conjuntos	47
20.3. Estructuras de datos	47
21.El módulo itertools	48
21.1. El módulo itertools	48
21.2. Iteradores infinitos	48
21.3. Operaciones sobre iterables	48
21.4. Funciones de combinatoria	48
22.Programación orientada a objetos	49
22.1. Programación orientada a objetos	49
22.2. Clases	49
22.3. Método __init__	49
22.4. Atributos	50
22.5. Métodos	50
22.6. Atributos de clase	50
22.7. Excepciones de clases	50
22.8. Herencia	50
22.9. Función super()	50
22.10Métodos mágicos	51
22.11Sobrecarga de operadores aritméticos	51
22.12Sobrecarga de operadores de comparación	51
22.13Métodos mágicos de contenedores	51
22.14Ciclo de vida de un objeto	51

22.15Encapsulamiento	52
23.Expresiones regulares	53
24.Empaquetamiento	54
25.Interfaz gráfica	55
26.Algoritmos de ordenamiento	56
27.Algoritmos de búsqueda	57
28.Algoritmos de matrices	58
29.Implementación de estructuras de datos	59
30.La librería NumPy	60

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 línea.

```
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.

```
1 # 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.

```
1 print(1 + 1) # 2
2 print(5 - 2) # 3
3 print(2 * 2) # 4
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.

```
1 print((2 + 3) * 4)
2 # (2 + 3) * 4
3 # 5 * 4
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.

```
1 print(0.25) # 0.25
2 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.

```
1 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.

```
1 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 $2^5 = 32$.

```
1 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.

```
1 print(7 // 2) # 3
```

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

```
1 print(7 % 2) # 1
```

Esta operación es equivalente a 7 mód 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.

```
1 print(10 % 2) # 0, 10 es múltiplo de 2
2 print(15 % 3) # 0, 15 es múltiplo de 3
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 `''`.

```
1 print('texto')
2 print("texto")
```

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

```
1 # Strings no válidos
2 print('texto")
3 print("texto')
```

3.2. Cadena vacía

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

```
1 cadena_vacia1 = ''
2 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.

```

1 print('\') # '
2 print('\") # "
3 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.

```

1 print("This isn't spanish")
2 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.

```

1 print('Hola\nMundo')
2 # Hola
3 # Mundo

```

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

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

```

1 print('Hola\tmundo')
2 # 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.

```

1 print('12345\r67')
2 # 67

```

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

```
1 print('123\b45')
2 # 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`.

```
1 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`.

```
1 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.

```
1 print('\u00f1') # ñ
2 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 multilinea

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`.

```
1 print("""Esta es
2 una cadena de
3 caracteres
4 multilinea""")
5 # Esta es
6 # una cadena de
7 # caracteres
8 # multilinea
```

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 +.

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

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

```
1 # 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.

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

También se pueden combinar operaciones de multiplicación y concatenación.

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

Multiplicar por 0 genera un string vacío.

```
1 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

<code>continue</code>	<code>if</code>	<code>return</code>
<code>def</code>	<code>import</code>	<code>True</code>
<code>del</code>	<code>in</code>	<code>try</code>
<code>elif</code>	<code>is</code>	<code>while</code>
<code>else</code>	<code>lambda</code>	<code>with</code>
<code>except</code>	<code>None</code>	<code>yield</code>

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.

```
1 x = 6
2 y = 7
3 print(x + y) # 13
```

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

```
1 x = 3
2 y = 4
3 z = x + y # 7
4 z = z + 1 # 8
5 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.

```
1 x = input()
2 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.

```
1 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.

```
1 x = '7'  
2 print(int(x)) # 7
```

- `float()`: Convierte a un float.

```
1 num = 42  
2 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.

```
1 edad = 25  
2 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 `**`.

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

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

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

Capítulo 5

Declaraciones if

5.1. Booleanos

Otro tipo de dato en Python es el tipo booleano, el cual sólo puede tener 2 valores: True y False. Se pueden crear comparando valores, por ejemplo, usando el operador de igualdad ==.

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

5.2. Operadores de comparación

Los operadores de comparación usados en Python son los siguientes:

- ==: Se evalúa como True si ambos elementos son iguales.
- !=: Se evalúa como True si ambos elementos son distintos.
- >: Se evalúa como True si el primer elemento es mayor.
- <: Se evalúa como True si el primer elemento es menor.
- >=: Se evalúa como True si el primer elemento es mayor o igual al segundo.
- <=: 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:

También se conocen como operadores relacionales.

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

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

5.3. Declaración if

Las declaraciones if sirven para ejecutar código si se 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.

```
1 if 3 > 2:  
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.

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

```
1 x = input()
2 if x % 2 == 0:
3     print(str(x) + ' es par')
4 else:
5     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.

```
1 x = input("Ingrese un número: ")
2 if x == 1:
3     print("Uno")
4 elif x == 2:
5     print("Dos")
6 elif x == 3:
7     print("Tres")
8 else:
9     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.

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

El operador not sólo toma una condición y la invierte.

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.

```
1 print(not (False and False) or True)
2 # not (False and False) or True
3 # not False or True
4 # True
5
6 print(not (False and (True or False))) # True
7 # not (False and (True or False))
8 # not (False and True)
9 # not False
10 # 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.

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

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

```
1 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.

```
1 animales = ["Perro", "Gato"]
2 print(animales[0]) # Perro
3 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.

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

6.3. IndexError en listas

Intentar acceder a un índice no existente producirá un error.

```
1 elementos = [False, 1, 2.0, "tres", [4]]
2 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.

```
1 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.

```
1 lista = ["hola", "mundo", [1, 2, 3]]
2 print(lista[0]) # hola
3 print(lista[1]) # mundo
4 print(lista[2]) # [1, 2, 3]
5 print(lista[2][0]) # 1
6 print(lista[2][1]) # 2
7 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.

```
1 identidad = [[1, 0, 0],
2              [0, 1, 0],
3              [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.

```

1 letras = ['a', 'b', 'c',]
2 print(letras) # ['a', 'b', 'c']
3 letras = ['a', 'b', 'c',,,] # SyntaxError

```

6.7. Operaciones con listas

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

```

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

```

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

```

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

```

1 frutas = ["manzana", "naranja", "pomelo", "limón", "durazno", "damasco"]
2 print("manzana" in frutas) # True
3 print("frutilla" in frutas) # False
4 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.

```

1 frutas = ["manzana", "naranja", "pomelo", "limón", "durazno", "damasco"]
2 print("frutilla" not in frutas) # True
3 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.

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

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

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

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

```
1 letras = ['a', 'b', 'c', 'd', 'e', 'f']
2 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.

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

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

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

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

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

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

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

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

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

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

```
1 nums = [1, 10, 100, 5, 6, -11, -52, 78, 0, 25]
2 print(max(nums)) # 100
3 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.

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

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

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

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

```
1 nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
2 nums.reverse()
3 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 descendente, se debe entregar el parámetro `reverse=True`.

```

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

```

También funciona con listas compuestas de números.

```

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

```

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

```

1 letras = ['a', 'b', 'c']
2 letras.clear()
3 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.

```

1 a = ['x', 'y']
2 b = a
3 b[0] = 'z'
4 print(b) # ['z', 'y']
5 print(a) # ['z', 'y']

```

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

```

1 a = ['x', 'y']
2 b = a.copy()
3 b[0] = 'z'
4 print(b) # ['z', 'y']
5 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.

```
1 hola = "Hola mundo"
2 print(hola[0]) # H
3 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.

```
1 frase = "Hola mundo"
2 lista = list(frase)
3 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 " ".

```
1 frase = "Hola mundo"
2 lista = frase.split()
3 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.

```
1 parrafo = "Lorem ipsum dolor sit amet, consectetur adipiscing elit..."
2 lista = parrafo.split()
3 print(lista) # ['Lorem', 'ipsum', 'dolor', 'sit', 'amet,', 'consectetur', 'adipiscing', 'elit..']
4 lista = parrafo.split(',')
5 print(lista) # ['Lorem ipsum dolor sit amet', ' consectetur adipiscing elit...']
6 lista = parrafo.split('e')
7 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.

```
1 i = 1 # contador
2 while i <= 5:
3     print(i)
4     i = i + 1
5 # 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.

```
1 i = 1
2 while True:
3     print(i)
4     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.

```
1 i = 0
2 while True:
```

```

3     print(i)
4     i = i + 1
5
6     if i >= 5:
7         print("fin")
8         break
9 # muestra los números del 0 al 4, seguido de "fin"

```

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

```

1 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.

```

1 print("Números del 1 al 10, exceptuando el 5:")
2 i = 0
3 while i < 10:
4     i = i + 1
5     if i == 5:
6         continue
7     print(i)

```

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

```

1 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.

También se puede usar para iterar sobre cadenas.

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.

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.

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.

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).

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.

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.

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:

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

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:

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`.

Cualquier código luego de la sentencia `return` nunca ocurrirá.

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.

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

Una forma de accederlo es usando la variable `__doc__`.

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

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

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.

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

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.

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.

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.

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.

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.

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.

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

9.2. Error de importación

Importar un módulo que no está disponible dará un `ImportError`.

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.

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

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:

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`.

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

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`.

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.

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.

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.

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.

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

12.4. Levantar excepciones

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

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

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

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, una excepción `AssertionError` es levantada.

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

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.

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.

Capítulo 13

Pruebas unitarias

Capítulo 14

Manejo de 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()`.

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:

14.3. 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.

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.

14.4. Lectura de archivos

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

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.

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.

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

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

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.

Esto también se puede hacer usando un bucle `for`.

14.5. Escritura de archivos

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

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

Para evitar que esto ocurra, se puede usar el modo anexo `"a"`.

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

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

14.6. 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`.

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`.

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:

Capítulo 15

Módulos time y datetime

Capítulo 16

Iterables

16.1. 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.

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

Capítulo 17

Diccionarios y tuplas

17.1. 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.

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.

Un diccionario puede almacenar como valor cualquier tipo de datos.

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

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`.

17.2. 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.

17.3. 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.

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

17.4. 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).

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).

17.5. Función keys()

17.6. 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.

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

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

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

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

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

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

Capítulo 18

Manejo de iterables

18.1. 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.

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.

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

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.

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

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

18.2. Cortes de tuplas

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

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.

18.3. 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.

18.4. Listas por compresión

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

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.

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.

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

18.5. 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.

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.

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

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.

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

18.6. 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:

18.7. Funciones numéricas

18.8. Funciones `all()` y `any()`

18.9. Función `enumerate()`

Capítulo 19

Programación funcional

19.1. 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.

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”.

19.2. 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:

Ejemplo de función impura:

19.3. 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.

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.

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

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

19.4. 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.

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

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

19.5. Función `filter()`

La función `filter()` es otra función de orden superior 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).

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

19.6. 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`.

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!

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.

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

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.

19.7. 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.

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.

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

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.

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

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

19.8. 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.

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.

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.

Otro ejemplo clásico es la serie de Fibonacci.

19.9. Iteración vs. Recursión

Capítulo 20

Conjuntos y estructuras de datos

20.1. Conjuntos

Los conjuntos son estructuras de datos parecidas a las listas o a los diccionarios. Son creados 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.

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

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.

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.

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

20.2. Operaciones con conjuntos

20.3. Estructuras de datos

Capítulo 21

El módulo itertools

21.1. El módulo itertools

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

21.2. Iteradores infinitos

21.3. Operaciones sobre iterables

21.4. Funciones de combinatoria

Capítulo 22

Programación orientada a objetos

22.1. 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.

22.2. 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.

22.3. 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__`.

22.4. Atributos

22.5. 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.

22.6. 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.

22.7. 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.

22.8. 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 sobrescribe.

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.

22.9. 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.

22.10. 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.

22.11. Sobrecarga de operadores aritméticos

22.12. 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.

22.13. 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.

22.14. 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.

22.15. 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 23

Expresiones regulares

Capítulo 24

Empaquetamiento

Capítulo 25

Interfaz gráfica

Capítulo 26

Algoritmos de ordenamiento

Capítulo 27

Algoritmos de búsqueda

Capítulo 28

Algoritmos de matrices

Capítulo 29

Implementación de estructuras de datos

Capítulo 30

La librería NumPy