



Resumen Python 3

Víctor Mardones Bravo

Índice general

| | |
|---|-----------|
| 1. Introducción a Python | 1 |
| 1.1. ¿Qué es Python? | 1 |
| 1.2. Aplicaciones | 1 |
| 1.3. Componentes principales | 2 |
| 1.4. Características de Python | 2 |
| 1.5. Organizaciones que lo usan | 2 |
| 1.6. Ventajas y desventajas | 3 |
| 1.7. Sintaxis | 3 |
| 1.8. Instalación | 4 |
| 1.9. Hola mundo | 5 |
| 1.10. El Zen de Python | 5 |
| 2. Números | 7 |
| 2.1. Comentarios | 7 |
| 2.2. Números enteros | 7 |
| 2.3. Operaciones aritméticas | 8 |
| 2.4. La regla PEMDAS | 9 |
| 2.5. Paréntesis | 9 |
| 2.6. Floats | 10 |
| 2.7. Exponenciación | 11 |
| 2.8. Cociente y resto | 12 |
| 3. Cadenas de caracteres | 13 |
| 3.1. Cadenas de caracteres | 13 |
| 3.2. Cadena vacía | 14 |
| 3.3. Caracteres especiales | 14 |
| 3.4. Secuencias de escape | 14 |
| 3.5. Caracteres Unicode | 16 |
| 3.6. Cadenas puras | 16 |
| 3.7. Strings multilínea | 16 |
| 3.8. Concatenación de strings | 17 |
| 3.9. Multiplicación de strings | 17 |
| 3.10. Subcadenas | 18 |
| 3.11. Opciones del método print() | 18 |
| 3.12. Párrafos | 20 |
| 4. Variables | 21 |
| 4.1. Asignación de variables | 21 |
| 4.2. Nombre de variables válidos | 21 |
| 4.3. Nombres largos de variables | 22 |
| 4.4. Palabras clave | 23 |
| 4.5. Operaciones con variables | 23 |
| 4.6. Entrada | 25 |
| 4.7. Conversión de tipos de datos | 25 |
| 4.8. Operadores de asignación | 27 |
| 4.9. Múltiples variables con el mismo valor | 27 |
| 4.10. Tipos de datos | 28 |
| 5. Declaraciones if y lógica | 31 |
| 5.1. Booleanos | 31 |

| | | |
|-----------|--|-----------|
| 5.2. | Conversión a booleanos | 32 |
| 5.3. | Operadores de comparación | 33 |
| 5.4. | Comparación entre strings | 34 |
| 5.5. | Declaración if | 35 |
| 5.6. | Indentación | 35 |
| 5.7. | Anidación de declaraciones if | 36 |
| 5.8. | Declaración if-else | 37 |
| 5.9. | Declaración elif | 37 |
| 5.10. | Operadores lógicos | 38 |
| 5.11. | Precedencia de operadores lógicos | 39 |
| 6. | Listas | 41 |
| 6.1. | Creación de listas | 41 |
| 6.2. | Índices | 41 |
| 6.3. | IndexError en listas | 42 |
| 6.4. | Lista vacía | 42 |
| 6.5. | Anidación de listas | 42 |
| 6.6. | Comas sobrantes | 43 |
| 6.7. | Tamaño de una lista | 43 |
| 6.8. | Operaciones con listas | 44 |
| 6.9. | Operadores in y not in | 45 |
| 6.10. | Añadir elementos a listas | 45 |
| 6.11. | Quitar elementos de listas | 46 |
| 6.12. | La declaración del | 48 |
| 6.13. | Ordenar listas | 48 |
| 6.14. | Información sobre listas | 49 |
| 6.15. | Copiar listas | 51 |
| 6.16. | Strings como listas | 52 |
| 6.17. | Conversión de strings a listas | 52 |
| 6.18. | Indexación de cadenas | 53 |
| 6.19. | Índices negativos | 53 |
| 6.20. | Reasignación de strings | 53 |
| 7. | Bucles | 55 |
| 7.1. | Bucles while | 55 |
| 7.2. | Bucles infinitos | 56 |
| 7.3. | Interrupción de entrada | 56 |
| 7.4. | Declaración break | 56 |
| 7.5. | Declaración continue | 57 |
| 7.6. | Bucle for con listas | 58 |
| 7.7. | Rangos | 59 |
| 7.8. | Rangos imposibles | 60 |
| 7.9. | Bucle for en rangos | 60 |
| 8. | Funciones | 62 |
| 8.1. | Reutilización de código | 62 |
| 8.2. | Funciones | 62 |
| 8.3. | Definición de funciones | 62 |
| 8.4. | Argumentos | 63 |
| 8.5. | Llamado de funciones | 63 |
| 8.6. | Variables globales y locales | 64 |
| 8.7. | Devolución de valores en una función | 65 |
| 8.8. | Docstring | 66 |
| 8.9. | Revisar documentación | 67 |
| 8.10. | Funciones como objetos | 67 |
| 8.11. | El objeto None | 68 |
| 8.12. | Sobrecarga de funciones | 69 |
| 8.13. | Anotaciones de tipos | 70 |
| 9. | Módulos y la biblioteca estándar | 71 |
| 9.1. | Módulos | 71 |

| | | |
|------------|--|------------|
| 9.2. | Error de importación | 72 |
| 9.3. | Alias | 72 |
| 9.4. | La biblioteca estándar | 73 |
| 9.5. | Módulos externos y pip | 74 |
| 9.6. | Obtener ayuda | 74 |
| 10. | El módulo math | 77 |
| 10.1. | El módulo math | 77 |
| 10.2. | Constantes matemáticas | 77 |
| 10.3. | Infinito | 78 |
| 10.4. | NaN | 79 |
| 10.5. | Funciones matemáticas predefinidas | 79 |
| 10.6. | Funciones matemáticas | 79 |
| 10.7. | Funciones de redondeo | 81 |
| 10.8. | Grados y radianes | 81 |
| 10.9. | Funciones trigonométricas | 82 |
| 10.10. | Números finitos e infinitos | 83 |
| 10.11. | Números complejos | 83 |
| 11. | El módulo random | 84 |
| 11.1. | El módulo random | 84 |
| 11.2. | La función random() | 84 |
| 11.3. | Generación de números enteros | 85 |
| 11.4. | Generación de números racionales | 86 |
| 11.5. | Semillas | 86 |
| 11.6. | Generación de listas aleatorias | 89 |
| 11.7. | Elementos al azar de secuencias | 89 |
| 11.8. | Distribuciones de probabilidad | 93 |
| 12. | Excepciones | 95 |
| 12.1. | Excepciones | 95 |
| 12.2. | Declaración try-except | 95 |
| 12.3. | Declaración finally | 97 |
| 12.4. | Levantar excepciones | 98 |
| 12.5. | Aserciones | 99 |
| 12.6. | Validación de entrada | 100 |
| 13. | Archivos | 103 |
| 13.1. | Abrir archivos | 103 |
| 13.2. | Modos de apertura | 103 |
| 13.3. | Extensión de modos de apertura | 105 |
| 13.4. | Cierre de archivos | 105 |
| 13.5. | Lectura de archivos | 105 |
| 13.6. | Escritura de archivos | 108 |
| 13.7. | Declaración with | 110 |
| 14. | Módulos time y datetime | 111 |
| 15. | Estructuras de datos | 112 |
| 15.1. | Estructuras de datos | 112 |
| 15.2. | Diccionarios | 112 |
| 15.3. | Listas como diccionarios | 112 |
| 15.4. | Diccionario vacío | 113 |
| 15.5. | Excepciones en diccionarios | 113 |
| 15.6. | Indexación de diccionarios | 114 |
| 15.7. | Uso de in y not en diccionarios | 114 |
| 15.8. | Función get() | 114 |
| 15.9. | Convertir diccionarios a listas | 115 |
| 15.10. | Tuplas | 116 |
| 15.11. | Indexación de Tuplas | 116 |
| 15.12. | Excepciones en tuplas | 116 |
| 15.13. | Anidación de tuplas | 117 |

| | |
|--|------------|
| 15.14. Tupla vacía | 117 |
| 15.15. Manejo de variables con tuplas | 117 |
| 15.16. Conjuntos | 118 |
| 15.17. Conjunto vacío | 118 |
| 15.18. Métodos de conjuntos | 119 |
| 15.19. Operaciones con conjuntos | 119 |
| 15.20. Frozenset | 120 |
| 15.21. Resumen sobre estructuras de datos | 122 |
| 16. Iterables | 123 |
| 16.1. Cortes de lista | 123 |
| 16.2. Cortes de tuplas | 124 |
| 16.3. Listas por comprensión | 125 |
| 16.4. Formateo de cadenas | 125 |
| 16.5. Funciones de cadenas | 126 |
| 16.6. Funciones all() y any() | 129 |
| 16.7. Función enumerate() | 129 |
| 17. Programación funcional | 131 |
| 17.1. Paradigma de programación funcional | 131 |
| 17.2. Funciones puras | 131 |
| 17.3. Lambdas | 132 |
| 17.4. Función map() | 133 |
| 17.5. Función filter() | 134 |
| 17.6. Generadores | 134 |
| 17.7. Decoradores | 136 |
| 17.8. Recursión | 139 |
| 17.9. Iteración vs. Recursión | 141 |
| 17.10. Generadores recursivos | 141 |
| 18. El módulo itertools | 142 |
| 18.1. El módulo itertools | 142 |
| 18.2. Iteradores infinitos | 142 |
| 18.3. Operaciones sobre iterables | 145 |
| 18.4. Funciones de combinatoria | 146 |
| 19. Programación orientada a objetos | 148 |
| 19.1. Programación orientada a objetos | 148 |
| 19.2. Objetos | 148 |
| 19.3. Clases | 148 |
| 19.4. Constructor | 149 |
| 19.5. Atributos y self | 150 |
| 19.6. Métodos | 151 |
| 19.7. Atributos de clase | 151 |
| 19.8. Excepciones de clases | 153 |
| 19.9. Herencia | 153 |
| 19.10. Función super() | 156 |
| 19.11. Métodos mágicos | 157 |
| 19.12. Sobrecarga de operadores aritméticos | 157 |
| 19.13. Métodos mágicos reversos | 159 |
| 19.14. Sobrecarga de operadores de comparación | 159 |
| 19.15. Métodos mágicos de contenedores | 160 |
| 19.16. Ciclo de vida de un objeto | 161 |
| 19.17. Encapsulamiento | 162 |
| 19.18. Métodos y atributos privados | 162 |
| 19.19. Métodos de clase | 163 |
| 19.20. Métodos estáticos | 164 |
| 19.21. Propiedades | 164 |
| 20. Expresiones regulares | 167 |
| 20.1. Expresiones regulares | 167 |

| | | |
|------------|---|------------|
| 20.2. | Definición de expresiones regulares | 167 |
| 20.3. | Coincidencias parciales | 168 |
| 20.4. | Coincidencias múltiples | 169 |
| 20.5. | Detalles sobre los resultados | 170 |
| 20.6. | Buscar y reemplazar | 170 |
| 20.7. | Coincidencias exactas | 171 |
| 20.8. | Metacaracteres | 171 |
| 20.9. | Punto | 172 |
| 20.10. | Inicio y final | 173 |
| 20.11. | Clases de caracteres | 174 |
| 20.12. | Rangos de caracteres | 175 |
| 20.13. | Clases invertidas | 177 |
| 20.14. | Repeticiones | 177 |
| 20.15. | Grupos de caracteres | 181 |
| 21. | PEP | 183 |
| 21.1. | PEP | 183 |
| 21.2. | PEP 8 | 183 |
| 21.3. | PEP 257 | 184 |
| 22. | Empaquetamiento | 185 |
| 23. | Pruebas unitarias | 186 |
| 23.1. | Pruebas unitarias | 186 |
| 23.2. | Precedencia de operadores completa | 186 |
| 24. | Interfaz gráfica | 187 |
| 25. | Algoritmos de ordenamiento | 188 |
| 25.1. | Bubble sort | 188 |
| 25.2. | Optimizaciones de bubble sort | 189 |
| 25.3. | Insertion sort | 190 |
| 25.4. | Selection sort | 192 |
| 25.5. | Merge sort | 193 |
| 25.6. | Quick sort | 195 |
| 25.7. | Heap sort | 195 |
| 26. | Algoritmos de búsqueda | 196 |
| 26.1. | Búsqueda lineal | 196 |
| 26.2. | Búsqueda binaria | 196 |
| 27. | Algoritmos de matrices | 197 |
| 28. | Implementación de estructuras de datos | 198 |
| 29. | La librería NumPy | 199 |

Capítulo 1

Introducción a Python

1.1. ¿Qué es Python?

Python es un lenguaje de programación popular y de alto nivel. Fue creado por Guido van Rossum y fue lanzado en el año 1991. Se han lanzado diferentes versiones hasta llegar a la versión actual, Python 3.

Según su creador, el nombre de Python viene de la serie de comedia británica “El Circo Volador de Monty Python”.

1.2. Aplicaciones

Python tiene aplicaciones en numerosas áreas y generalmente se usa para:

- Scripting
- Desarrollo web
- Desarrollo de software
- Prototipos rápidos
- Desarrollo de aplicaciones para empresas
- Redes y seguridad
- Automatización de tareas
- Programación de bots
- Cálculos complejos
- Estudios científicos y computación científica
- Desarrollo de juegos
- Desarrollo de aplicaciones móviles
- Extracción de datos de sitios web
- Análisis de datos
- Inteligencia artificial y modelos de machine learning
- Procesamiento de imágenes
- Internet de las cosas

1.3. Componentes principales

- **Funciones:** Python tiene funciones predefinidas que son de gran utilidad, por ejemplo las funciones matemáticas. Las funciones son grupos de bloques de código que pueden ejecutarse en cualquier sección del código, según el programador lo necesite.
- **Clases:** Son los planos para construir objetos, los cuales incluyen todas sus características y comportamientos. En Python, prácticamente todo es un objeto, aunque en algunos casos no parezca obvio.
- **Módulos:** Agrupan funciones y clases que sirven para objetivos similares. La biblioteca estándar de Python es muy grande comparada con las de otros lenguajes de programación.
- **Paquetes:** Agrupan módulos usados en aplicaciones grandes, permitiendo su distribución a terceros.

Si estos conceptos no se entienden todavía, no hay problema, pues se verán más a fondo en sus respectivos capítulos.

1.4. Características de Python

- **Soporte de múltiples plataformas:** Python es “platform independent”. Esto significa que un mismo código puede ejecutarse en cualquier sistema operativo, ya sea Windows, Unix, Linux o Mac, sin necesidad de hacerle cambios.
- **Interpretado:** El código escrito en Python no necesita ser compilado, como ocurre en lenguajes como C, Java o C++. El intérprete de Python convierte el código en bytes ejecutables línea por línea, lo cual hace el desarrollo de prototipos más conveniente para el programador, pero al mismo tiempo hace que la ejecución del código sea más lenta.
- **Simple:** La sintaxis de Python es simple y fácil de leer, aunque requiere un conocimiento de inglés básico. Esta simplicidad también permite escribir código más corto, obteniendo el mismo resultado en menos líneas que en otros lenguajes.
- **Robusto:** Esto significa que el lenguaje es capaz de manejar los posibles errores que podrían ocurrir durante la ejecución de programas escritos en él.
- **De alto nivel:** Es un lenguaje de nivel usado para scripting. Esto quiere decir que el programador no necesita recordar características de bajo nivel como la arquitectura del sistema o el manejo de memoria. Estas características se abstraen.
- **Multiparadigma:** Python soporta programación estructurada, funcional y orientada a objetos.
- **Gran soporte de librerías:** Python puede integrarse a otras librerías que tengan funcionalidades específicas. No hay necesidad de tener que escribir el código por tu cuenta, si ya existe y funciona correctamente.
- **Integrable:** El código fuente escrito en Python puede usarse dentro de otros lenguajes de programación, lo cual permite expandir las funcionalidades de sus programas con las de programas escritos en otros lenguajes.
- **De código abierto:** Python es de [código abierto](#). Se puede usar sin necesidad de tomar su licencias y se puede descargar fácilmente.
- **Gratuito:** Ninguna persona u organización necesita pagar para usarlo en sus proyectos.
- **Conciso y compacto:** El código escrito en Python es conciso y compacto, lo que sirve para que los programadores puedan entenderlo rápidamente.
- **Dinámicamente tipado:** El tipo de dato de cada variable se decide durante el tiempo de ejecución, lo que significa que no es necesario declarar su tipo en el código.

1.5. Organizaciones que lo usan

Python es muy popular y es usado por organizaciones y aplicaciones como:

- [Microsoft](#)
- [Google](#)

- [Yahoo](#)
- [Mozilla](#)
- [Cisco](#)
- [Facebook](#)
- [Spotify](#)
- [OpenStack](#)
- [NASA](#)
- [CIA](#)
- [Disney](#)

1.6. Ventajas y desventajas

Ventajas:

- Es de código abierto y fácil de empezar a usar.
- Es fácil de aprender y explorar.
- Se pueden integrar módulos de terceros con facilidad.
- Es un lenguaje de alto nivel y orientado a objetos.
- Es interactivo y portable.
- Las aplicaciones pueden ejecutarse en cualquier plataforma.
- Es dinámicamente tipeado.
- Tiene una comunidad grande y foros activos.
- Tiene una sintaxis intuitiva.
- Tiene librerías con amplio soporte.
- Es un lenguaje interpretado.
- Se puede conectar con bases de datos.
- Aumenta la productividad debido a su simplicidad.

Desventajas:

- No puede usarse para desarrollo de aplicaciones móviles.
- Su acceso a bases de datos es limitado.
- Consume más memoria por ser dinámicamente tipeado.
- Su ejecución es lenta comparada con lenguajes compilados.
- Las aplicaciones y código requieren mayor mantenimiento, debido a su gran grado de abstracción.
- Debido a su abstracción, algunos conceptos de programación importantes podrían ser omitidos durante su aprendizaje.

1.7. Sintaxis

Python fue diseñado para ser fácil de leer y tiene similitudes con el idioma inglés, con influencia de las matemáticas.

A diferencia de otros lenguajes de programación, Python termina las líneas con un salto de línea, en vez de un punto y coma `;`.

Para delimitar bloques de código en estructuras de control como `if`, `while`, `for` o funciones, se utiliza indentación con espacios en blanco en vez de usar llaves `{ }` como en otros lenguajes.

1.8. Instalación

Muchos PCs vienen con Python preinstalado. Para revisar la versión que se tiene instalada se puede ingresar el siguiente comando en el terminal.

```
python --version
```

Salida:

```
Python 3.9.2
```

Siempre se puede descargar la última versión gratuitamente desde el [sitio web oficial](#). Después de instalarlo, existen muchas formas de utilizar este lenguaje.

Se puede abrir la consola de Python en el terminal ingresando el siguiente comando.

```
python
```

Salida:

```
Python 3.9.2 (tags/v3.9.2:1a79785, Feb 19 2021, 13:44:55) [MSC v.1928 64 bit (AMD64)]
> on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Si no funciona, existe otro comando que también puede abrir la consola.

```
py
```

Dentro de la consola, se puede ingresar el código línea por línea, y es interpretado después de pulsar **Intro**. Para salir de la consola, se debe llamar a la función `exit()`.

```
exit()
```

Para ejecutar un archivo de extensión `.py`, se puede ingresar `python` seguido de la ruta relativa (desde el directorio de trabajo) o absoluta del archivo.

```
python archivo.py
```

También existen muchos IDEs (Integrated Development Environment) que permiten programar en Python. Estos contienen muchas herramientas que pueden ser de utilidad.

Algunos IDEs y editores de texto recomendados son los siguientes:

- IDLE, que viene preinstalado con Python.
- [Thonny](#)
- [Pycharm](#)
- [Visual Studio Code](#), que requiere instalar [esta extensión](#).
- [Sublime Text](#)

1.9. Hola mundo

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

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

Salida:

```
Hola mundo
```

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

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

Salida:

```
Hola  
Mundo
```

Si no se le entrega ningún texto a `print()`, se imprimirá una línea vacía. El texto puede escribirse entre comillas simples o dobles.

```
print("Hola")  
print()  
print("Mundo")
```

Salida:

```
Hola  
  
Mundo
```

1.10. El Zen de Python

El [Zen de Python](#) es una colección de 20 “principios”, 19 de ellos escritos por Tim Peters. Estos principios han servido como inspiración para muchos programadores de todo el mundo a la hora de crear software.

Se mostrará en pantalla como un “huevo de pascua” la primera vez que se ejecute la siguiente línea, para importar el módulo `this`.

```
import this
```

Después se mostrará el siguiente texto, en inglés.

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!

Una traducción posible sería la siguiente:

El Zen de Python, escrito por Tim Peters

Bello es mejor que feo.
Explícito es mejor que implícito.
Simple es mejor que complejo.
Complejo es mejor que complicado.
Plano es mejor que anidado.
Espaciado es mejor que denso.
La legibilidad es importante.
Los casos especiales no son lo suficientemente especiales como para romper las reglas.
Sin embargo la practicidad le gana a la pureza.
Los errores nunca deberían pasar silenciosamente.
A menos que se silencien explícitamente.
Frente a la ambigüedad, evitar la tentación de adivinar.
Debería haber una, y preferiblemente solo una, manera obvia de hacerlo.
A pesar de que esa manera no sea obvia a menos que seas Holandés.
Ahora es mejor que nunca.
A pesar de que nunca es muchas veces mejor que *ahora* mismo.
Si la implementación es difícil de explicar, es una mala idea.
Si la implementación es fácil de explicar, puede que sea una buena idea.
Los espacios de nombres son una gran idea, itengamos más de esos!

Nota: Si se observa con claridad, se puede ver que el principio 20 “no existe”. Tim Peters, quien lo escribió, dejó este último principio para que Guido (el creador de Python) lo llene. El origen de este texto puede encontrarse [en esta conversación](#).

Capítulo 2

Números

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
```

La mayoría de editores de código marcan los comentarios con su propio color, distinto al del resto de código.

```
# Este es un comentario
print('Hola mundo') # Este es otro comentario
# Otro comentario más
```

Salida:

```
Hola mundo
```

Python no tiene comentarios multilínea para fines generales como lo tienen otros lenguajes de programación tales como C. Para comentar varias líneas de código se debe antener un comentario a cada línea.

```
# Python no soporta
# comentarios multilínea.
# Esta es la
# única opción
# en esos casos.
```

Por último, se debe recordar no abusar del uso de comentarios. No es conveniente llenar el código de comentarios o comentar cosas que son demasiado obvias.

```
# Ejemplo de comentario innecesario:
# Muestra el texto "hola mundo" en pantalla
print('hola mundo')
```

2.2. Números enteros

Los números enteros (integer) son el tipo de dato más básico en muchos lenguajes de programación, y Python no es la excepción.

Para mostrar un número en pantalla, sólo basta con usar la función `print()` y entregarle el número. El número debe escribirse sin usar comillas y Python inferirá su tipo automáticamente.

```
# positivo
print(5)
print(10)

# neutro
print(0)

# negativo
print(-10)
```

Salida:

$$\begin{array}{r} 5 \\ 10 \\ 0 \\ -10 \end{array}$$

A diferencia de otros lenguajes de programación, los números en Python 3 son de largo arbitrario, en otras palabras, soportan cualquier cantidad de dígitos. Esto hace que los cálculos con números grandes sean mucho más convenientes, pero reduce el rendimiento.

```
print(9999999999999999999999999999999)
```

Salida:

99999999999999999999999999999999

2.3. Operaciones aritméticas

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

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

Salida:

2
3
4
1.0

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

El resultado de una división es un número decimal. Esto lo convierte al tipo de dato float, el cual se verá más tarde en este capítulo.

```
print(5 / 2)
```

Salida:

2.5

Todas las operaciones se pueden combinar entre sí como si el intérprete de Python fuera una calculadora.

```
print(1 + 2 - 3 + 10 - 7)
```

Salida:

3

2.4. La regla PEMDAS

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

- 1) **Paréntesis** `()`
- 2) **Exponentes** `**`
- 3) **Multiplicación** `*` y **División** `/` (de izquierda a derecha)
- 4) **Adición** `+` y **Sustracción** `-` (de izquierda a derecha)

A continuación se muestra un ejemplo detallando el orden en el que se procesan las operaciones.

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

Salida:

19.0

En realidad, la regla PEMDAS es parte de una jerarquía de operadores mucho más grande, que incluye todos los operadores que se pueden usar en Python. Esta jerarquía se verá más tarde.

2.5. Paréntesis

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

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

Salida:

20

2.6. Floats

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

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

Salida:

```
0.25
0.75
```

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

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

Salida:

```
0.30000000000000004
```

El error mostrado arriba es un error clásico de la aritmética de punto flotante. Es un error tan conocido que incluso tiene su propio [sitio web](#).

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

```
print(.42)
```

Salida:

```
0.42
```

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

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

```
# entero y float
print(1 + 2.0)
print(15.0 - 7)
print(3 * 4.0)
print(10.0 / 2)

# enteros
print(10 / 2)
```

Salida:

```
3.0
8.0
12.0
5.0
5.0
```

Las operaciones entre floats y enteros son posibles porque Python convierte los enteros en floats silenciosamente al momento de realizarlas.

2.8. Cociente y resto

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

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

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

Salida:

```
3
```

También se puede usar la división entera con floats, lo cual dará como resultado otro float.

```
print(6.25 // 0.5)
```

Salida:

```
12.0
```

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

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

Salida:

```
1
```

Esta operación es equivalente a 7 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.

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

Salida:

```
0
0
2
```

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

El operador módulo `%` también puede usarse con floats.

```
print(6.25 % 0.5)
print(2.5 % 1.25)
```

Salida:

```
0.25
0.0
```

Capítulo 3

Cadenas de caracteres

3.1. Cadenas de caracteres

Las cadenas de caracteres, también conocidas en inglés como strings, se crean introduciendo el texto entre comillas simples `' '` o dobles `" "`.

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

Salida:

```
texto
texto
```

El uso de comillas simples o dobles no afecta en ninguna forma el comportamiento del string, es decir, ambos producen el mismo resultado.

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

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

Salida:

```
File "<stdin>", line 2
  print('texto")
                ^
SyntaxError: EOL while scanning string literal
```

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

Salida:

```
File "<stdin>", line 2
  print("texto')
                ^
SyntaxError: EOL while scanning string literal
```

3.2. Cadena vacía

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

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

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

3.3. Caracteres especiales

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

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

Salida:

```
'
"
\
```

Los caracteres `'`, `"` y `\` son especiales, porque normalmente cumplen funciones especiales dentro de strings.

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

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

Salida:

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

3.4. Secuencias de escape

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

Las secuencias de escape más usadas son:

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

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

Salida:

```
Hola
Mundo
```

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

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

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

Salida:

| | | | |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
|---|---|---|---|

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

Otras secuencias de escape que no son tan usadas son:

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

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

Salida:

| |
|-------|
| 67345 |
|-------|

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

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

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

Salida:

| |
|------|
| 1245 |
|------|

- **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”.

El resultado obtenido de estas últimas puede variar dependiendo del IDE que se utilice.

3.5. Caracteres Unicode

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

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

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

Salida:

```
ñ
ñ
```

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

3.6. Cadenas puras

Las cadenas puras “raw string” ignoran las secuencias de escape, como `\n` o `\t`. Se escriben igual que una cadena común, pero con la letra `r` fuera de las comillas.

```
x = r"holaN\tmundo"
print(x)

y = "holaN\nmundo"
print(y)
```

Salida:

```
holaN\tmundo
holaN
mundo
```

En ocasiones su uso puede ser conveniente.

3.7. 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`.

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

Salida:

```
Esta es
una cadena de
caracteres
multilinea
```

3.8. Concatenación de strings

Se pueden realizar operaciones matemáticas no solo con números, sino también con cadenas.

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

```
print("Hola" + "mundo") # sin espacio
print("Hola" + " " + "mundo") # con espacio
print("1" + "1")
```

Salida:

```
Holamundo
Hola mundo
11
```

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

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

Salida:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    print(1 + "1")
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

3.9. Multiplicación de strings

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

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

Salida:

```
jajajaja
jajajaja
```

También se pueden combinar operaciones de multiplicación y concatenación.

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

Multiplicar por `0` genera un string vacío.

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

Salida:

Las cadenas no pueden ser multiplicadas entre sí, tampoco pueden ser multiplicadas por floats, incluso si los floats son números enteros.

```
print('foo' * 'bar')
```

Salida:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    print('foo' * 'bar')
TypeError: can't multiply sequence by non-int of type 'str'
```

```
print('hola' * 3.0)
```

Salida:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    print('hola' * 3.0)
TypeError: can't multiply sequence by non-int of type 'float'
```

3.10. Subcadenas

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

```
cadena = "Hola Python"

print(cadena[:4])
print(cadena[5:])
print(cadena[1:4])
```

Salida:

```
Hola
Python
ola
```

3.11. Opciones del método print()

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

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

Salida:

```
a b c
```


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

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

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

Salida:

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

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

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

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

Salida:

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

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

Salida:

```
a b c.
x
```

Ambos argumentos se pueden combinar.

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

Salida:

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

3.12. Párrafos

Una secuencia de escape útil para escribir párrafos o cualquier texto largo en general, que no tenga saltos de línea en su interior, es usando la secuencia de escape `\` seguida de un salto de línea (en el código).

Esto permite escribir strings largos que no pueden verse completamente en pantalla sin tener que moverse hacia la derecha. El uso de esta secuencia de escape no altera el resultado.

```
print("Lorem ipsum dolor sit amet, consectetur adipiscing elit, \
sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. \
Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris \
nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in \
reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla \
pariatur. Excepteur sint occaecat cupidatat non proident, sunt in \
culpa qui officia deserunt mollit anim id est laborum.")
```

Salida:

```

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
↳  incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud
↳  exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute
↳  irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla
↳  pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui
↳  officia deserunt mollit anim id est laborum.
```

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 de igualdad =.

```
usuario = "Juan"

print(usuario)
```

Salida:

```
usuario = "Juan"

print(usuario)
```

En Python, no existe la declaración de variables. Toda variable que se cree debe tener un valor inicial asignado. Se permite que este valor pueda ser “vacío”, como por ejemplo el string vacío "".

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.

No seguir estas reglas hará que ocurran errores que impiden la ejecución del programa.

```
123 = 44
```

Salida:

```
File "<stdin>", line 1
  123 = 44
    ^
SyntaxError: cannot assign to literal
```

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

Salida:

```
File "<stdin>", line 1
  123variable = 'hola'
    ^
SyntaxError: invalid syntax
```

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

Salida:

```
File "<stdin>", line 1
  hola mundo = 'hola mundo'
    ^
SyntaxError: invalid syntax
```

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

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

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

Salida:

```
10
3
5
```

4.3. Nombres largos de variables

Los nombres de variables que se componen de más de una palabra pueden ser difíciles de leer. Existen muchas técnicas para escribirlos de forma más legible.

- **Camel Case:** Todas las palabras empiezan con mayúscula, excepto la primera.

```
variableDeNombreLargo = 42
```

- **Pascal Case:** Todas las palabras empiezan con mayúscula.

```
VariableDeNombreLargo = 42
```

- **Snake Case:** Todas las palabras se escriben en minúscula, pero van separadas por guiones bajos `_`.

```
variable_de_nombre_largo = 42
```

4.4. 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**.

4.5. Operaciones con variables

Se pueden usar variables dentro de operaciones.

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

Salida:

```
13
```

Pero se debe recordar que deben ser asignadas antes de poder usarlas.

```
print(x + y)
```

Salida:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    print(x + y)
NameError: name 'x' is not defined
```

Una variable también puede cambiar de valor a lo largo de la ejecución de un programa, y ser el resultado de operaciones con otras variables.

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

Durante el tiempo de ejecución, siempre mantendrá el último valor que se le asignó.

```
x = 5
print(x + 3)
print(x)

x = 10
print(x + 3)
print(x)
```

Salida:

```
8
5
13
10
```

Las variables se pueden reasignar tantas veces como se desee, para cambiar su valor. En Python, las variables no tienen tipos específicos, por lo que se puede asignar una cadena a una variable y luego asignar un número entero a la misma variable.

```
x = 12.5
print(x)

x = "Cadena de caracteres"
print(x)
```

Salida:

```
12.5
Cadena de caracteres
```

Sin embargo, no es una buena práctica. Para evitar errores, se debería evitar sobrescribir una variable con distintos tipos de datos.

Si se asigna el valor numérico (enteros y decimales) o alfanumérico (cadena de texto) de una variable a otra, los cambios hechos a ambas variables son independientes.

```
x = 2
print(x)

y = x
print(y)

x = 3
print(x)
print(y)
```

Salida:

```
2
2
3
2
```

Esto no ocurre para tipos de datos más complejos, como listas.

4.6. Entrada

Para obtener información del usuario, se puede usar la función `input()`, la cual solicita al usuario que la entrada. La información obtenida puede ser almacenada como una variable.

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

Entrada:

5

Salida:

5

Toda la información recibida por el método `input()` se procesa 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.

Se puede usar `input()` varias veces para tomar múltiples entradas del usuario.

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

print(nombre + " " + apellido)
```

Entrada:

Fulanito
Pérez

Salida:

Fulanito Pérez

Normalmente, se debe añadir un espacio para concatenar variables, para evitar que las variables aparezcan juntas en la salida.

4.7. Conversión de tipos de datos

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

- `int()`: Convierte a un número entero.

```
x = '7'

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

Salida:

```
7
7
```

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

```
x = '7'

print(x + '10')
print(int(x) + 10)
```

Salida:

```
710
17
```

Al convertir de **float** a **int**, sólo se conservará la parte entera del número. No ocurrirá ningún tipo de redondeo.

```
x = 12.55
print(int(x))

y = -9.55
print(int(y))
```

Salida:

```
12
-9
```

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

```
num = 42

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

Salida:

```
42
42.0
```

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

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

Salida:

Edad: 25

Como el resultado de la función `input()` es un string, es normal convertir el resultado de este método para usarlo en operaciones matemáticas.

```
x = float(input("Ingrese un número: "))  
  
print(x + 10)
```

Entrada:

7

Salida:

17.0

También se pueden usar estos métodos para forzar variables a ser de un tipo específico al momento de asignarlas, en vez de dejar que Python infiera su tipo. Esto se conoce como “casting”.

```
x = str(3) # x será '3'  
y = int(3) # y será 3  
z = float(3) # z será 3.0
```

4.8. 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
```

4.9. Múltiples variables con el mismo valor

Para declarar muchas variables en una misma línea y hacer que todas tengan el mismo valor, se puede repetir el operador de asignación `=`.

```
x = y = z = 1  
  
print(x)  
print(y)  
print(z)
```

Salida:

```
1  
1  
1
```

4.10. Tipos de datos

En Python existen muchos tipos de datos que pueden ser usados y almacenados como variables. La función `type()` retorna el tipo de dato del objeto que se le entrega como argumento.

A continuación se mostrarán los tipos de datos más comunes.

- Número entero `int`

```
n = 10  
print(type(n))
```

Salida:

```
<class 'int'>
```

- Número racional `float`

```
num = 12.5  
print(type(num))
```

Salida:

```
<class 'float'>
```

- Cadena de caracteres `str`

```
mes = "Febrero"  
print(type(mes))
```

Salida:

```
<class 'str'>
```

- Booleano `bool`

```
es_azul = True
print(type(es_azul))
```

Salida:

```
<class 'bool'>
```

- Lista **list**

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

Salida:

```
<class 'list'>
```

- Rango **range**

```
rango = range(0, 5)
print(type(rango))
```

Salida:

```
<class 'range'>
```

- Función **function**

```
def saludar():
    print("hola mundo")
print(type(saludar))
```

Salida:

```
<class 'function'>
```

- Número complejo **complex**

```
z = 2 + 3j
print(type(z))
```

Salida:

```
<class 'complex'>
```

- Diccionario **dict**

```
vocales = {'A': 'a', 'E': 'e', 'I': 'i', 'O': 'o', 'U': 'u'}  
print(type(vocales))
```

Salida:

```
<class 'dict'>
```

- Tupla **tuple**

```
numeros = (1, 2, 3)  
print(type(numeros))
```

Salida:

```
<class 'tuple'>
```

- Conjunto **set**

```
frutas = {"manzana", "pera", "piña"}  
print(type(frutas))
```

Salida:

```
<class 'set'>
```

- Frozenset **frozenset**

```
colores = frozenset({"rojo", "verde", "azul"})  
print(type(colores))
```

Salida:

```
<class 'frozenset'>
```

Estos tipos y muchos más se verán en detalle en los próximos capítulos.

Capítulo 5

Declaraciones if y lógica

5.1. Booleanos

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

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

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

Salida:

```
True
False
```

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

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

Salida:

```
True
False
True
```

Se debe tener cuidado de no confundir el operador de asignación `=` con el operador de igualdad `==`.

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

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

Salida:

```
True
True
```

Usando otros términos, podría decirse que **1** es “truthy” y que **0** es “falsy”. Esta equivalencia también es válida para floats.

```
print(1.0 == True)
print(0.0 == False)
```

Salida:

```
True
True
```

5.2. Conversión a booleans

La función `bool()` se puede usar para convertir un objeto a su valor booleano.

Al usarla con valores numéricos, el número `0` se convierte a `False` y cualquier otro número se convierte a `True`.

```
print(bool(0))
print(bool(0.0))

print(bool(1))
print(bool(1.0))

print(bool(-10))
print(bool(2.5))
```

Salida:

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

Una cadena de texto vacía se convierte en `False` y cualquier cadena que no sea vacía se convierte en `True`.

```
print(bool(""))
print(bool("spam"))
print(bool("eggs"))
```

Salida:

```
False
True
True
```

La misma situación ocurre al convertir tipos de datos más complejos, los cuales no se han visto todavía.

```
# lista
print(bool([]))
print(bool([1, 2, 3]))

# conjunto
print(bool({}))
print(bool({1, 2, 3}))

# tupla
print(bool(()))
print(bool((1, 2, 3)))
```

Salida:

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

5.3. Operadores de comparación

Los operadores de comparación usados en Python son los siguientes:

- **Igual** `=`, escrito como `==`: Se evalúa como **True** si ambos elementos son iguales.
- **Distinto** `≠`, escrito como `!=`: Se evalúa como **True** si ambos elementos son distintos.
- **Mayor que** `>`, escrito como `>`: Se evalúa como **True** si el primer elemento es mayor.
- **Menor que** `<`, escrito como `<`: Se evalúa como **True** si el primer elemento es menor.
- **Mayor o igual que** `≥`, escrito como `>=`: Se evalúa como **True** si el primer elemento es mayor o igual al segundo.
- **Menor o igual que** `≤`, escrito como `<=`: Se evalúa como **True** si el primer elemento es menor o igual al segundo.

En los casos contrarios, los operadores retornan **False**.

Ejemplos de uso de operadores de comparación:

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

Salida:

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

También se conocen como operadores relacionales.

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

```
print(1 == 1.0)
```

Salida:

```
True
```

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

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

Salida:

```
False
False
False
```

5.4. Comparación entre strings

Los operadores `==` y `!=` funcionan entre strings como se esperaría.

```
print("agua" == "aceite")
print("huevo" == "huevo")
```

Salida:

```
False
True
```

Los operadores `>`, `<`, `>=` y `<=` también se pueden usar para comparar cadenas lexicográficamente (por orden alfabético), donde las palabras que están después se consideran “mayores”.

```
print("a" < "d")
print("b" > "c")
print("Ana" > "Alicia")
```

Salida:

```
True
False
True
```

Las letras mayúsculas siempre irán antes (son “menores”) que las minúsculas al ordenarlas alfabéticamente.

```
print('A' == 'a')
print('B' < 'b')
print('a' > 'Z')
```


Salida:

```
False
True
True
```

Específicamente, Python compara los códigos [ASCII](#) para ordenar las cadenas de textos.

5.5. 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.

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

Salida:

```
3 es mayor que 2
```

5.6. Indentación

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 `{}`.

Para que el intérprete reconozca los bloques de código correctamente, cada instrucción dentro de un mismo bloque de código debe escribirse a un mismo nivel de indentación.

A continuación se muestra un ejemplo que no cumple con eso. Para que se entienda el concepto mejor, en cada segmento de código en esta sección se mostrarán los espacios en blanco.

```
if (10 > -10):
    print("hola")
    print("mundo")
    print("chao")
```

Salida:

```
File "<stdin>", line 3
    print("mundo")
      ^
IndentationError: unindent does not match any outer indentation level
```

No se permite mezclar espacios con saltos de tabulador (los que aparecen al pulsar la tecla `Tab`). Aunque se vean igual, dependiendo de las configuraciones, un salto de tabulador puede cambiar de medida. La medida de un espacio es siempre la misma, así que se recomienda usarlos en vez del tabulador.

A continuación se muestra un ejemplo que parece ser válido a simple vista, pero no lo es ya que tiene un salto de tabulador en la línea `print("mundo")` y 4 espacios en las otras líneas.

```
if (10 > -10):
    print("hola")
    →print("mundo")
    print("chao")
```

Salida:

```
File "<stdin>", line 3
    print("mundo")
TabError: inconsistent use of tabs and spaces in indentation
```

Pára evitar que esto ocurra, muchos IDEs tienen la opción “insertar espacios al presionar la tecla **Tab**”, para insertar 4 espacios (o la cantidad que se desee) al pulsar dicha tecla.

Por último, aunque esto no es un error de por sí (no se lanza ninguna excepción), se recomienda mantener la cantidad de espacios usados para cada indentación consistente, es decir, que no varíe su cantidad.

El siguiente ejemplo muestra una práctica que se debe evitar.

```
#_4_espacios
if_(1==_1):
    _print("foo")
    _print("bar")

#_2_espacios
if_(2==_2):
    _print("spam")
    _print("eggs")

#_8_espacios
if_(3==_3):
    _print("hola")
    _print("mundo")
```

Salida:

```
foo
bar
spam
eggs
hola
mundo
```

No existe una cantidad “correcta” de espacios para indentar, aunque la mayoría de programadores prefiere que sean 2, 3 o 4 (en Python). Lo más importante es que se escoja una cantidad y se mantenga igual dentro de un mismo proyecto.

Los errores de indentación son más fáciles de evitar que de encontrar, así que como dice el dicho, “mejor prevenir que lamentar”.

5.7. Anidación de declaraciones if

Para realizar comprobaciones más complejas, las declaraciones **if** se pueden anidar, una dentro de la otra. Esto significa que la declaración **if** interna sólo se evalúa si la declaración **if** externa es válida.

```
num = 15

if num > 10:
    print("Mayor que 10")

    if num < 30:
        print("Menor que 30")

        if num > 20:
            print("Mayor que 20")
```

Salida:

```
Mayor que 10
Menor que 30
```

5.8. 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))
```

Salida:

```
6 es mayor que 5
```

```
x = int(input())

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

Entrada:

```
3
```

Salida:

```
3 es impar
```

Entrada:

```
3
```

Salida:

```
3 es impar
```

5.9. Declaración elif

Cada declaración `if` sólo puede tener una declaración `else` que la acompañe, lo cual puede resultar muy incómodo.

```

numero = 3

if numero == 1:
    print("Uno")
else:
    if numero == 2:
        print("Dos")
    else:
        if numero == 3:
            print("Tres")
        else:
            print("Otro número")

```

Salida:

Tres

El término **elif** se usa para evitar tener que anidar múltiples declaraciones **if-else**. Hace que el código sea más corto.

```

x = int(input("Ingrese un número: "))

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

```

Entrada:

2

Salida:

Dos

Entrada:

10

Salida:

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**.

5.10. Operadores lógicos

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

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

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

Salida:

```
True
False
```

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

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

Salida:

```
True
True
```

El operador **not** sólo toma una condición y la invierte.

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

Salida:

```
True
```

5.11. Precedencia de operadores lógicos

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

- 1) Paréntesis **()**
- 2) Operador “no” **not**
- 3) Operador “y” **and**
- 4) Operador “o” **or**

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

```
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
```

Salida:

True

Capítulo 6

Listas

6.1. Creación de listas

Las listas son la estructura más usada en Python. Son colecciones ordenadas posicionalmente de objetos tipados arbitrariamente y no tienen un tamaño fijo. Se utilizan para almacenar varios elementos y se crean utilizando corchetes y separando los elementos por comas.

```
animales = ['perro', 'gato', 'paloma']  
primos = [2, 3, 5, 7, 11]
```

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

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

6.2. Índices

Se puede acceder a un determinado elemento de la lista escribiendo su índice entre corchetes `[]`. El índice del primer elemento es `0`, en lugar de `1`, como podría esperarse. Después del `0` viene el `1`, luego el `2`, y así sucesivamente.

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

Salida:

```
Perro  
Gato
```

Alternativamente, se pueden usar índices reversos. El último elemento de una lista tiene índice `-1`, el anterior `-2`, y así sucesivamente 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
```

Salida:

```
León  
Mono  
Gato  
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])
```

Salida:

```
Traceback (most recent call last):  
  File "<stdin>", line 2, in <module>  
    print(elementos[5])  
IndexError: list index out of range
```

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.

```
anidados = [[0,1], [2,3], [4,5]]  
  
# Elementos de la lista más grande  
print(anidados[0])  
print(anidados[1])  
print(anidados[2])  
  
# Elementos de las listas más pequeñas  
print(anidados[0][0])  
print(anidados[0][1])  
  
print(anidados[1][0])  
print(anidados[1][1])  
  
print(anidados[2][0])  
print(anidados[2][1])
```

Salida:


```
[0, 1]
[2, 3]
[4, 5]
0
1
2
3
4
5
```

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](#).

```
# la alineación es para que se vea más parecido a una matriz
identidad = [[1, 0, 0],
             [0, 1, 0],
             [0, 0, 1]]
```

El intérprete ignora los saltos de línea y la indentación dentro de los corchetes `[]`, lo que permite ordenarla para hacer más fácil su comprensión.

Esta estructura tipo matriz es muy conveniente en los casos en los que se necesite almacenar datos en formato tabla (fila-columna).

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)
```

Salida:

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

Si se deja más de una coma, se mostrará un `SyntaxError`.

```
letras = ['a', 'b', 'c',,,]
```

Salida:

```
File "<stdin>", line 1
  letras = ['a', 'b', 'c',,,]
              ^
SyntaxError: invalid syntax
```

6.7. Tamaño de una lista

Se puede obtener el tamaño (número de elementos) de una lista utilizando la función `len()`.

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

numeros = [10, 11, 12]
print(len(numeros))
```

Salida:

```
6
3
```

6.8. 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)
```

Salida:

```
[0, 0, 1, 0]
```

Sin embargo, no se puede usar este método para añadir elementos a una lista. Para hacer eso, se deben usar funciones que se verán más adelante.

```
vocales = ['a', 'e', 'i', 'o']

vocales[4] = 'u'
```

Salida:

```
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
    vocales[4] = 'u'
IndexError: list assignment index out of range
```

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

```
nums = [1, 2, 3]

print(nums + [4, 5, 6])
print(nums * 4)
```

Salida:

```
[1, 2, 3, 4, 5, 6]
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
```

6.9. 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)
print("frutilla" in frutas)
print("durazno" in frutas)
```

Salida:

```
True
False
True
```

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

```
frutas = ["manzana", "naranja", "pomelo", "limón", "durazno", "damasco"]

print("frutilla" not in frutas)
print(not "frutilla" in frutas)
```

Salida:

```
True
True
```

6.10. Añadir elementos a listas

El método `append()` permite añadir un elemento al final de una lista.

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

print(lista)
```

Salida:

```
[1, 2, 3, 4, 5, 6]
```

El método `insert()` se usa para insertar un elemento en un índice específico de una lista. Toma 2 argumentos: un índice y un objeto. Los elementos que están después del elemento insertado se desplazan a la derecha.

```
palabras = ["adiós", "bye"]

palabras.insert(1, "ciao")
print(palabras)

palabras.insert(0, "au revoir")
print(palabras)
```

Salida:

```
['adiós', 'ciao', 'bye']  
['au revoir', 'adiós', 'ciao', 'bye']
```

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) # se extendió  
print(nums2) # sin cambios
```

Salida:

```
[1, 2, 3, 4, 5, 6]  
[4, 5, 6]
```

El resultado obtenido es el mismo que al concatenar las listas, pero la diferencia es que reemplaza la primera lista por la versión extendida.

6.11. Quitar elementos de listas

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

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

Salida:

```
[1, 2, 3, 4]
```

Se le puede entregar un argumento numérico para quitar elementos de una lista según su índice. Este número puede ser negativo.

```
lista = ['a', 'b', 'c', 'd', 'e', 'f']  
  
lista.pop(2)  
print(lista)  
  
lista.pop(2)  
print(lista)  
  
lista.pop(0)  
print(lista)  
  
lista.pop(-1)  
print(lista)
```

Salida:

```
['a', 'b', 'd', 'e', 'f']  
['a', 'b', 'e', 'f']  
['b', 'e', 'f']  
['b', 'e']
```

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)
```

Salida:

```
[1, 2, 1, 2, 3]
```

Si el elemento no existe en la lista, levanta un `ValueError`.

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

Salida:

```
Traceback (most recent call last):  
  File "<stdin>", line 2, in <module>  
    numeros.remove(4)  
ValueError: list.remove(x): x not in list
```

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

```
numeros = [1, 2, 3, 1, 2, 3]  
  
# evita que ocurra una excepción  
if 4 in numeros:  
    numeros.remove(4)  
  
print(numeros)
```

Salida:

```
[1, 2, 3, 1, 2, 3]
```

El método `clear()` quita todos los elementos de una lista, dejándola vacía.

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

Salida:

```
[]
```

6.12. La declaración del

Otra forma de eliminar elementos de una lista es usando la palabra clave **del**. Esto permite eliminar elementos según su índice, similar a usar **pop()**.

```
alumnos = ['Mario', 'María']  
  
del alumnos[0] # índice  
  
print(alumnos)
```

Salida:

```
['María']
```

También se puede especificar un rango de eliminación, lo que hace que elimine elementos desde el índice antes de **:** hasta el antecesor del índice después de **:**.

```
nombres = ['Pedro', 'Yolanda', 'Javiera', 'Óscar', 'Diego', 'Víctor', 'Alberto',  
→ 'Verónica', 'Teresa', 'Héctor', 'Olivia', 'Nicolás']  
  
del nombres [2:8]  
  
print(nombres)
```

Salida:

```
['Pedro', 'Yolanda', 'Teresa', 'Héctor', 'Olivia', 'Nicolás']
```

6.13. Ordenar listas

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**.

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

Salida:

```
['a', 'c', 'd', 'f', 'h', 'i', 'j', 'm', 'x', 'y', 'z']  
['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)
```

Salida:

```
[-100, -99, -22, -1, 0, 1, 5, 25, 37, 99]
```

Si se usa en listas mixtas, mostrará un **TypeError**.

```
mezcla = ['z', 100, 'a', 0, 'j', -1, 'd', 24, 'h', 52, 'c']  
mezcla.sort()
```

Salida:

```
Traceback (most recent call last):  
  File "<stdin>", line 3, in <module>  
    mezcla.sort()  
TypeError: '<' not supported between instances of 'int' and 'str'
```

El método **reverse()** invierte el orden de los elementos de una lista.

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

Salida:

```
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

6.14. Información sobre listas

Existen métodos que no modifican listas, pero pueden obtener información sobre ellas.

El método **index()** encuentra la primera aparición de un elemento en una lista y devuelve su índice.

```
letras = ['a', 'z', 'x', 'b', 'd', 'h', 'k']  
  
print(letras.index('x'))  
print(letras.index('d'))
```

Salida:

```
2  
4
```

Si el elemento no se encuentra en la lista, muestra un **ValueError**.

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

Salida:

```
Traceback (most recent call last):  
  File "<stdin>", line 3, in <module>  
    print(letras.index('m'))  
ValueError: 'm' is not in list
```

También acepta un segundo argumento, donde se puede especificar el índice desde el cual se empezará a buscar.

```
abc = ['a', 'b', 'c', 'a', 'b', 'c', 'a', 'b', 'c']  
  
print(abc.index('a'))  
print(abc.index('b', 3))  
print(abc.index('c', 6))
```

Salida:

```
0  
4  
8
```

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))  
print(lista.count(1))
```

Salida:

```
7  
5
```

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))  
print(min(nums))
```

Salida:

```
100  
-52
```

Si son strings, los compara por orden alfabético (lexicográficamente). 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))  
print(min(palabras))
```

Salida:

```
variable  
casa
```

En realidad, se comparan los símbolos según su valor ASCII.

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


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

Salida:

```
21
```

Se pueden combinar las funciones `avg()` y `len()` para obtener el promedio de los elementos de una lista numérica.

```
lista = [5, 6, 10]
promedio = sum(lista) / len(lista)
print(promedio)
```

Salida:

```
lista = [5, 6, 10]
promedio = sum(lista) / len(lista)
print(promedio)
```

6.15. 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)
print(a)
```

Salida:

```
['z', 'y']
['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)
print(a)
```

Salida:

```
['z', 'y']
['x', 'y']
```

Otra opción válida es usando el constructor `list()`.

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

print(b)
print(a)
```

Salida:

```
['z', 'y']
['x', 'y']
```

6.16. Strings como listas

Las cadenas pueden verse como listas de caracteres que no se pueden cambiar (inmutables).

```
# Se parecen bastante, ¿no?
cadena = "Python"
lista = ['P', 'y', 't', 'h', 'o', 'n']
```

De hecho, tienen muchas similitudes. Ambas almacenan elementos secuencialmente. La diferencia es que las cadenas son más limitadas (pero también más rápidas) porque sólo pueden almacenar caracteres y son inmutables, lo que significa que sus contenidos no se pueden alterar.

6.17. 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)
```

Salida:

```
['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']
```

Salida:

```
['Hola', 'mundo']
```

6.18. Indexación de cadenas

Algunos tipos de variables, como las cadenas, se pueden indexar. Esto se realiza de una forma parecida a las listas.

```
hola = "Hola mundo"

print(hola[0])
print(hola[1])
print(hola[4]) # un espacio
print(hola[5])
```

Salida:

```
H
o

m
```

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

6.19. Índices negativos

En las cadenas, se pueden obtener sus caracteres empezando desde el final de la palabra. Para hacer esto, se debe hacer la indexación normal, pero usando índices negativos. El último carácter tiene índice **-1**, el anterior **-2**, y así sucesivamente.

```
hola = "¡Hola mundo!"

print(hola[-1])
print(hola[-2])
print(hola[-3])
```

Salida:

```
!
o
d
```

Esto resulta especialmente útil con cadenas largas.

6.20. Reasignación de strings

A diferencia de las listas, las cadenas son inmutables, lo que significa que no se pueden reasignar sus caracteres.

```
cadena = "lana"
cadena[1] = "u"
```

Salida:

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    cadena[1] = "u"
TypeError: 'str' object does not support item assignment
```

La única forma de solucionar este problema es creando una nueva cadena con el cambio que se quería.

```
cadena = "lana"  
cadena = "luna"  
  
print(cadena)
```

Salida:

```
luna
```

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
```

Salida:

```
1
2
3
4
5
```

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

Se puede realizar cualquier operación con el contador dentro del bucle.

```
i = 2
while i <= 100:
    print(i)
    i *= 2
```

Salida:

```
2
4
8
16
32
64
```

```
i = 2
while i <= 1000:
    print(i)
    i **= 2
```

Salida:

```
2
4
16
256
```

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
```

Salida:

```
1
2
3
4
5
6
7
.
.
.
```

7.3. Interrupción de entrada

En caso de querer detener la ejecución del programa, por ejemplo, cuando ocurre un bucle infinito, se puede interrumpir su ejecución en cualquier momento enviando un **KeyboardInterrupt** al pulsar la combinación de teclas **Ctrl+C**.

Salida:

```
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
    print(i)
KeyboardInterrupt
```

En el ejemplo de arriba se muestra lo que ocurre al interrumpir el bucle **while** infinito de la sección anterior.

7.4. 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
```

Salida:

```
0
1
2
3
4
fin
```

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

`break`

Salida:

```
File "<stdin>", line 1
  break
  ^
SyntaxError: 'break' outside loop
```

7.5. 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.

```
i = 0
while i < 7:
    i = i + 1

    if i == 4:
        continue
    print(i)
```

Salida:

```
1
2
3
5
6
7
```

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

`continue`

Salida:

```
File "<stdin>", line 1
  continue
  ^
SyntaxError: 'continue' not properly in loop
```

7.6. Bucle for con listas

Otro tipo de bucle es el bucle **for**. Estos bucles se pueden usar para iterar sobre una lista. Las instrucciones dentro de su bloque de código se ejecutarán para cada elemento de la lista.

```
animales = ["perro", "gato", "jirafa", "elefante", "mono"]

for animal in animales:
    print(animal)
```

Salida:

```
perro
gato
jirafa
elefante
mono
```

También se puede usar para iterar sobre cadenas.

```
texto = "probando bucles for"

for caracter in texto:
    print(caracter)
```

Salida:

```
p
r
o
b
a
n
d
o

b
u
c
l
e
s

f
o
r
```

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


```

numeros = [1, 9, 8, 2, 1, 0, 0, 2, 4, 0, 3, 5, 6, 8]
for n in numeros:

    if n == 0:
        continue
    if n == 3:
        break

    print(n)
# muestra los números de la lista, omitiendo los 0 y termina cuando llega al 3

```

Salida:

```

1
9
8
2
1
2
4

```

7.7. Rangos

La función `range()` devuelve una secuencia de números.

Si se llama a `range()` con un argumento, produce un objeto con valores desde `0` hasta el antecesor del número especificado.

```

numeros = range(10)
print(numeros)

```

Salida:

```
range(0, 10)
```

Para mostrar los números dentro de un objeto `range`, debe convertirse a una lista usando el método `list()`.

```

numeros = list(range(10))
print(numeros)

```

Salida:

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

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

```

numeros = list(range(3, 8))
print(numeros)

```

Salida:

```
[3, 4, 5, 6, 7]
```

Si el primer número es **0**, el resultado es equivalente al de ingresar sólo 1 argumento.

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

Salida:

```
True
```

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

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

print(nums1)
print(nums2)
```

Salida:

```
[0, 2, 4, 6, 8, 10, 12, 14]
[15, 13, 11, 9, 7, 5, 3, 1]
```

7.8. Rangos imposibles

Intentar almacenar rangos “imposibles” no mostrará ningún error. El rango simplemente no tendrá ningún elemento, lo cual es más fácil de ver al convertirlo en una lista.

```
# es imposible ir de 0 a 10 dando saltos de -1 en -1
nums = range(0, 10, -1)

print(nums)
print(list(nums))
```

Salida:

```
range(0, 10, -1)
[]
```

7.9. Bucle for en rangos

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

```
for i in range(6):
    print(i)
```

Salida:

```
0
1
2
3
4
5
```

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.

En general, un bucle `for` puede recorrer cualquier iterable. Estos se verán más a fondo en su propio capítulo.

Capítulo 8

Funciones

8.1. Reutilización de código

La reutilización de código es una parte muy importante de la programación en cualquier lenguaje. Incrementar el tamaño del código lo hace más difícil de mantener.

Para que un proyecto grande de programación sea exitoso, es esencial que se atenga al principio DRY, del inglés Don't Repeat Yourself (no te repitas).

Del código malo y repetitivo, se dice que se atiene al principio WET, del inglés Write Everything Twice (escribe todo dos veces) o We Enjoy Typing (disfrutamos escribir).

Los bucles vistos en el capítulo anterior son una forma de reutilizar código. Este capítulo trata sobre funciones, que son otra forma de hacer esta práctica.

8.2. Funciones

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

Ejemplos de funciones que se han visto anteriormente:

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

Palabras antes del paréntesis son nombres de funciones y los valores separados por comas dentro de los paréntesis son argumentos o parámetros de funciones.

8.3. 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.

Generalmente, se recomienda que el nombre de una función sea un verbo, pero no es obligatorio.

Ejemplo de definición de función:

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

8.4. Argumentos

Las funciones más básicas son las que no reciben argumentos, pero esto limita su utilidad en gran cantidad. Para hacerlas más útiles, se les puede entregar argumentos, lo que aumenta su funcionalidad.

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

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

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

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

8.5. 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:

```
# definición  
def saludar():  
    print("Hola")  
  
# llamada  
saludar()
```

Salida:

```
Hola
```

```
# definición  
def exclamar(palabra):  
    print("!" + palabra + "!")  
  
# llamadas  
exclamar("Hola mundo")  
exclamar("Miau")  
exclamar("Guau")
```

Salida:

```
!Hola mundo!  
!Miau!  
!Guau!
```

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

# llamadas
sumar(3, 2)
sumar(-10, 10)
sumar(5.7, 25.14)
```

Salida:

```
5
0
30.84
```

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

# llamadas
mayor(100, 1000)
mayor(-1, 1)
mayor(2.5, 2.6)
```

Salida:

```
1000
1
2.6
```

Técnicamente, los parámetros son las variables en una definición de función y los argumentos son los valores que se le dan a las variables al momento de llamar funciones.

8.6. Variables globales y locales

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

```
def funcion(variable):
    variable += 1
    print(variable)

funcion(6)
print(variable)
```

Salida:

```
7
Traceback (most recent call last):
  File "<stdin>", line 6, in <module>
    print(variable)
NameError: name 'variable' is not defined
```

En términos más técnicos, una variable que se inicia fuera de una función recibe el nombre de variable global y una que se declara dentro de una función se conoce como variable local.

```
# variable global, puede usarse en todo el código
x = 1

def funcion():
    # variable local, sólo se puede usar dentro de la función
    y = 2
```

Los parámetros de una función también son variables locales.

Existe una excepción a esta regla, la cual permite declarar variables globales dentro del cuerpo de una función. Para hacer esto, se debe usar la palabra clave **global**.

```
x = 1 # variable global

def funcion():
    y = 2 # variable local

    global z # declaración de variable global
    z = 3 # asignación de variable global

funcion()
print(z)
```

Salida:

```
3
```

Es importante darse cuenta de que la función **funcion()** debe ser llamada para inicializar la variable global **z** y que al usar la palabra clave **global** se debe separar la declaración de la asignación.

Si no se separan, se mostrará un **SyntaxError**.

```
def funcion():
    global z = 3
```

Salida:

```
File "<stdin>", line 2
    global z = 3
            ^
SyntaxError: invalid syntax
```

8.7. Devolución de valores en una función

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

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

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

Salida:

```
9
```

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

```
def sumar(x, y):  
    print('Esto si ocurrirá')  
    return x + y  
    print('Esto nunca ocurrirá')  
  
print(sumar(4, 5))
```

Salida:

```
Esto si ocurrirá  
9
```

8.8. 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 y teniendo cuidado de respetar la indentación.

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

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

```
def sumar(x, y):  
    """  
    Parámetros:  
    x: Primer número  
    y Segundo número  
    Retorna:  
    La suma de ambos números  
    """  
    return x + y
```

El intérprete de Python ignora los docstring que se usen en otros lugares. Esto ha hecho que algunas personas los usen como “comentarios multilínea”, pero en la práctica no se recomienda hacer esto porque puede generar confusión.

```
"""  
Este es un docstring,  
no un comentario multilínea.  
"""  
  
# Se recomienda usar  
# este formato.
```


También podría generar problemas de indentación en algunos casos.

8.9. Revisar documentación

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 durante el tiempo de ejecución.

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

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

Salida:

```
print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

Prints the values to a stream, or to sys.stdout by default.
Optional keyword arguments:
file: a file-like object (stream); defaults to the current sys.stdout.
sep:  string inserted between values, default a space.
end:  string appended after the last value, default a newline.
flush: whether to forcibly flush the stream.
```

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

```
help(print)
```

Salida:

```
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file: a file-like object (stream); defaults to the current sys.stdout.
    sep:  string inserted between values, default a space.
    end:  string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

8.10. Funciones como objetos

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

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

a = 4
b = 7
producto = multiplicar

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

Salida:

```
28
28
```

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

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

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

a = 5
b = 10

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

Salida:

```
30
```

8.11. El objeto None

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

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

Cuando es ingresado a la consola de Python, se visualiza como una cadena vacía. Usando **print()** se mostrará su verdadero valor.

```
None # cadena vacía
print(None)
print(None == None)
```

Salida:

```
None
True
```

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

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

var = saludar()
print(var)
```

Salida:

```
Hola mundo
None
```

Un uso común de **None** es para iniciar una lista con cierta cantidad de elementos.

```
lista = [None] * 10
print(lista)
```

Salida:

```
[None, None, None, None, None, None, None, None, None, None]
```

8.12. Sobrecarga de funciones

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

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

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

saludar()
saludar("Juan")
```

Salida:

```
Hola
Hola, Juan
```

El siguiente ejemplo muestra una función que calcula el área de una figura geométrica. Si sólo se le entrega 1 argumento asume que es un cuadrado y si se le entregan 2 asume que es un rectángulo.

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

print(area(4)) # Cuadrado
print(area(5, 6)) # Rectángulo
```

Salida:

```
16
30
```

8.13. 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.

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

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

Salida:

```
holahola
```

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

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

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

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

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

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

Salida:

```
15
17.6
ab
```

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

Capítulo 9

Módulos y la biblioteca estándar

9.1. Módulos

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

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

```
import random

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

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

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

```
from math import pi

print(pi)
```

Salida:

```
3.141592653589793
```

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

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

También se pueden importar todos los objetos de un módulo usando un `*`. Esto generalmente no se recomienda, ya que confunde las variables del código original con las del módulo externo.

```
from math import *

variable = 3

print(variable)
print(pi) # ya no necesita escribir math.pi, pero puede confundir
```

Salida:

```
3
3.141592653589793
```

Las variables que provienen de módulos también se pueden reasignar, aunque en la mayoría de casos no hay razón para hacerlo.

```
from math import pi

pi = 7
print(pi)
```

Salida:

```
7
```

Para ver una lista de módulos disponibles, se puede usar la función `help()` de la siguiente forma:

```
help('modules')
```

9.2. Error de importación

Importar un módulo que no existe dará un `ModuleNotFoundError`, o uno que no está disponible dará un `ImportError`.

```
import modulo_desconocido
```

Salida:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    import modulo_desconocido
ModuleNotFoundError: No module named 'modulo_desconocido'
```

9.3. Alias

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

```
from math import sqrt as raiz_cuadrada

print(raiz_cuadrada(42))
```

Salida:

```
6.48074069840786
```

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

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

print(raiz_cuadrada(numero))
```

Salida:

```
1.7724538509055159
```

Después de darle un alias a un objeto, su nombre original no funcionará.

```
from math import sqrt as raiz_cuadrada, pi as numero
print(sqrt(numero)) # no existe sqrt
```

Salida:

```
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
    print(sqrt(numero)) # no existe sqrt
NameError: name 'sqrt' is not defined
```

La notación mixta, importando algunos objetos con alias y otros sin alias, está permitida.

```
from math import sqrt as raiz_cuadrada, pi, e as euler
print(raiz_cuadrada(pi))
print(raiz_cuadrada(euler))
```

Salida:

```
1.7724538509055159
1.6487212707001282
```

Otra opción posible es importar un módulo con un nombre distinto.

```
import math as mates
print(mates.pi)
```

Salida:

```
3.141592653589793
```

9.4. La biblioteca estándar

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

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

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

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

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

La razón por la que algunos módulos fueron escritos en [C](#) fue porque necesitaban realizar acciones de más bajo nivel que lo posible usando únicamente Python.

9.5. Módulos externos y pip

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

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

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

```
pip install nombre_de_libreria
```

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

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

```
pip help
```

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

9.6. Obtener ayuda

Cuando se trabaja con Python, como cualquier otro lenguaje, es muy común olvidar los métodos que se pueden usar con determinados tipos de datos.

La función **dir()** es una función integrada en Python que permite ver los métodos que puede usar un determinado objeto.

```
nombre = 'Juan'

# Métodos de un string
print(dir(nombre))
```

Salida:


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

Esto permite saber con facilidad qué métodos se pueden usar con la variable que se está usando.

```
numero = 10

# Métodos de un entero
print(dir(numero))
```

Salida:

```
[ '__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__', '__delattr__',
↳ '__dir__', '__divmod__', '__doc__', '__eq__', '__float__', '__floor__',
↳ '__floordiv__', '__format__', '__ge__', '__getattr__', '__getnewargs__',
↳ '__gt__', '__hash__', '__index__', '__init__', '__init_subclass__', '__int__',
↳ '__invert__', '__le__', '__lshift__', '__lt__', '__mod__', '__mul__', '__ne__',
↳ '__neg__', '__new__', '__or__', '__pos__', '__pow__', '__radd__', '__rand__',
↳ '__rdivmod__', '__reduce__', '__reduce_ex__', '__repr__', '__rfloordiv__',
↳ '__rlshift__', '__rmod__', '__rmul__', '__ror__', '__round__', '__rpow__',
↳ '__rrshift__', '__rshift__', '__rsub__', '__rtruediv__', '__rxor__', '__setattr__',
↳ '__sizeof__', '__str__', '__sub__', '__subclasshook__', '__truediv__', '__trunc__',
↳ '__xor__', 'as_integer_ratio', 'bit_length', 'conjugate', 'denominator',
↳ 'from_bytes', 'imag', 'numerator', 'real', 'to_bytes']
```

Los métodos rodeados por `__` son métodos mágicos, se verán en el capítulo de programación orientada a objetos.

El resultado obtenido al usar `dir()` se puede combinar con el método `help()` para obtener más detalles sobre un método en específico.

```
nombre = 'Juan'

# Ver documentación sobre el método startswith() de un string
help(nombre.startswith)
```

Salida:

```
Help on built-in function startswith:

startswith(...) method of builtins.str instance
    S.startswith(prefix[, start[, end]]) -> bool

    Return True if S starts with the specified prefix, False otherwise.
    With optional start, test S beginning at that position.
    With optional end, stop comparing S at that position.
    prefix can also be a tuple of strings to try.
```

El método `help()` también se puede usar para obtener ayuda detallada sobre módulos importados.

```
import math

help(math)
```

Salida:

```
Help on built-in module math:

NAME
    math

DESCRIPTION
    This module provides access to the mathematical functions
    defined by the C standard.

FUNCTIONS
    acos(x, /)
        Return the arc cosine (measured in radians) of x.

        The result is between 0 and pi.

    acosh(x, /)
        Return the inverse hyperbolic cosine of x.

    asin(x, /)
        Return the arc sine (measured in radians) of x.

        The result is between -pi/2 and pi/2.

    asinh(x, /)
        Return the inverse hyperbolic sine of x.

    atan(x, /)
-- Más --
```

Por defecto, el método `help()` sólo mostrará el texto que alcance en el alto de la ventana del terminal. Al pulsar la tecla **Intro**, se mostrará la línea que viene después de `-- Más --`. Como siempre, se puede pulsar **Ctrl+C** para salir.

También se puede usar el método `dir()` en módulos, para ver los métodos dentro de ellos.

```
import math

print(dir(math))
```

Salida:

```
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh',
↳ 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'comb', 'copysign', 'cos',
↳ 'cosh', 'degrees', 'dist', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial',
↳ 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose',
↳ 'isfinite', 'isinf', 'isnan', 'isqrt', 'lcm', 'ldexp', 'lgamma', 'log', 'log10',
↳ 'log1p', 'log2', 'modf', 'nan', 'nextafter', 'perm', 'pi', 'pow', 'prod',
↳ 'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc',
↳ 'ulp']
```

Capítulo 10

El módulo math

10.1. El módulo math

El módulo `math` contiene funciones y constantes comúnmente usadas en matemáticas generales.

```
import math
```

En este capítulo también se verán otras funciones matemáticas que están disponibles en Python por sí solo, sin necesidad de importar este módulo.

10.2. Constantes matemáticas

El módulo `math` tiene 5 constantes matemáticas muy importantes. Primero, se verán las 3 constantes numéricas que almacenan los valores de números irracionales importantes.

La constante `pi` representa el número π , el cual es la relación entre el perímetro y diámetro de una circunferencia.

$\pi = 3,14159\dots$

```
import math  
print(math.pi)
```

Salida:

```
3.141592653589793
```

La constante `tau`, escrita en matemáticas como τ , es el doble de `pi`, y representa la relación entre el perímetro y el radio de una circunferencia.

$\tau = 2\pi$

```
import math  
print(math.tau)  
print(math.tau == 2 * math.pi)
```

Salida:

```
6.283185307179586  
True
```

La constante `e` representa el [número de Euler](#), otro de los números irracionales más importantes.

$e = 2,71828\dots$

```
import math
print(math.e)
```

Salida:

```
2.718281828459045
```

10.3. Infinito

Para representar un número [infinito](#), se puede usar el método `float()` y entregarle el string `"inf "` como argumento.

∞

```
print(float("inf")) # infinito positivo
print(float("-inf")) # infinito negativo
```

Salida:

```
inf
-inf
```

Una forma alternativa de hacerlo es usando la constante `inf` del módulo `math`.

```
import math
print(math.inf)
print(-math.inf)
```

Salida:

```
inf
-inf
```

Ambas formas de hacerlo entregan exactamente el mismo resultado.

```
import math
print(math.inf == float('inf'))
print(-math.inf == float('-inf'))
```

Salida:

```
True
True
```

10.4. NaN

La constante `nan` o `NaN`, acrónimo proveniente del inglés Not a Number (no es un número) no proviene directamente de las matemáticas, pero tiene usos bastante útiles en programación. Representa un número “ilegal”.

```
import math

print(math.nan)
```

Salida:

```
nan
```

Generalmente se usa para expresar resultados imposibles de calcular, como raíces negativas o indeterminaciones. Esta constante se almacena como un float.

```
import math

print(type(math.nan))
```

Salida:

```
<class 'float'>
```

10.5. Funciones matemáticas predefinidas

Python viene con algunas funciones matemáticas sencillas, que se pueden usar sin necesidad de importar el módulo `math`. En algunos casos, el módulo `math` provee versiones más completas de estas funciones. Para obtener la distancia entre un número y el 0 (su valor absoluto), puede usarse la función `abs()`.

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

Salida:

```
42
42
```

10.6. Funciones matemáticas

La función `fabs()` retorna el valor absoluto de un número, al igual que `abs()`, pero siempre retorna un float.

```
import math

print(math.fabs(-5))
print(abs(-5))
```

Salida:

```
5.0
5
```

Como ya se ha visto, la función `sqrt()`, abreviación del inglés square root, retorna la raíz cuadrada de un número positivo o `0`. El resultado obtenido siempre será la raíz positiva.

```
import math

print(math.sqrt(16))
print(math.sqrt(1.5))
print(math.sqrt(0))
print(math.sqrt(math.e))
```

Salida:

```
4.0
1.224744871391589
0.0
1.6487212707001282
```

No se puede usar con números negativos.

```
import math

print(math.sqrt(-1))
```

Salida:

```
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
    print(math.sqrt(-1))
ValueError: math domain error
```

La función `factorial()` retorna el factorial de un número.

```
import math

print(math.factorial(0)) # 1, por definición
print(math.factorial(1)) # 1
print(math.factorial(2)) # 2*1
print(math.factorial(3)) # 3*2*1
print(math.factorial(4)) # 4*3*2*1
print(math.factorial(5)) # 5*4*3*2*1
```

Salida:

```
1
1
2
6
24
120
```

El factorial no está definido para los números negativos.

```
import math

print(math.factorial(-1))
```

Salida:

```
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
    print(math.factorial(-1))
ValueError: factorial() not defined for negative values
```

10.7. Funciones de redondeo

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

```
from math import pi

print(round(pi, 0))
print(round(pi, 1))
print(round(pi, 2))
print(round(pi, 4))
print(round(pi, 8))
```

Salida:

```
3.0
3.1
3.14
3.1416
3.14159265
```

10.8. Grados y radianes

Al trabajar con ángulos, es común medirlos usando grados o radianes. El módulo `math` tiene funciones para convertir valores entre estas unidades.

El método `radians()` convierte de grados a radianes.

```
import math

print(math.radians(360)) # 2pi
print(math.radians(180)) # pi
print(math.radians(100.05))
print(math.radians(-20))
```

Salida:

```
6.283185307179586
3.141592653589793
1.7462019166203266
-0.3490658503988659
```

El método `degrees()` convierte de radianes a grados.

```
import math

print (math.degrees(math.pi))
print (math.degrees(math.tau))
print (math.degrees(0))
print (math.degrees(3 * math.pi / 4))
print (math.degrees(7 * math.pi / 4))
```

Salida:

```
180.0
360.0
0.0
135.0
315.0
```

10.9. Funciones trigonométricas

El módulo `math` tiene funciones trigonométricas, similares a las de una calculadora. Estas funciones sólo funcionan con radianes. Para usarlas con grados, primero deben convertirse a radianes con el método `radians()`.

Funciones trigonométricas básicas:

```
import math

print(math.sin(math.pi)) # seno
print(math.cos(0)) # coseno
print(math.tan(math.pi / 4)) # tangente

print(1 / math.cos(0)) # secante
print(1 / math.sin(math.pi)) # cosecante
print(1 / math.tan(math.pi / 4)) # cotangente
```

Salida:

```
1.2246467991473532e-16
1.0
0.9999999999999999
1.0
8165619676597685.0
1.0000000000000002
```

Funciones trigonométricas inversas:

```
import math

print(math.asin(1)) # arcoseno
print(math.acos(0)) # arcocoseno
print(math.atan(1)) # arcotangente

print(1 / math.acos(0)) # arcosecante
print(1 / math.asin(1)) # arcocosecante
print(1 / math.atan(1)) # arcocotangente
```

Salida:


```
1.5707963267948966
1.5707963267948966
0.7853981633974483
0.6366197723675814
0.6366197723675814
1.2732395447351628
```

Funciones trigonométricas hiperbólicas:

```
import math

print(math.sinh(math.pi)) # seno hiperbólico
print(math.cosh(0)) # coseno hiperbólico
print(math.tanh(math.pi / 4)) # tangente hiperbólica

print(1 / math.cosh(0)) # secante hiperbólica
print(1 / math.sinh(math.pi)) # cosecante hiperbólica
print(1 / math.tanh(math.pi / 4)) # cotangente hiperbólica
```

Salida:

```
11.548739357257746
1.0
0.6557942026326724
1.0
0.08658953753004696
1.5248686188220641
```

Se debe recordar que los resultados de estas funciones se obtienen por aproximaciones y un computador no puede representar todos los números decimales con precisión perfecta, lo que significa que puede haber pequeños errores de aproximación en los resultados.

10.10. Números finitos e infinitos

10.11. Números complejos

Capítulo 11

El módulo random

11.1. El módulo random

El módulo `random` pertenece a la biblioteca estándar y contiene funciones para generar números pseudoaleatorios.

```
import random
```

Los números generados son pseudoaleatorios porque la aleatoriedad “natural” no existe en los computadores. Todas las funciones que generan elementos al azar en realidad emplean algoritmos muy difíciles de calcular o de predecir en el poco tiempo que tardan en retornar un resultado.

La mayoría de ejemplos de código en este capítulo tienen resultados variables, es decir, cada vez que se ejecute el código los resultados serán distintos. La excepción a esta regla es cuando se especifica una semilla.

11.2. La función random()

La función `random()` genera un float aleatorio entre 0 y 1.

```
import random

print(random.random())
```

Salida:

```
0.35770526186133567
```

Para mostrar su funcionamiento mejor, se puede usar dentro de un bucle.

```
import random

for i in range(10):
    print(random.random())
```

Salida:

```
0.5428710873911465
0.5382815653766047
0.3427394612983208
0.9870199209838638
0.2345661096878311
0.9860180454282067
0.4900049868582307
0.4276553050524561
0.763671330679946
0.2379499909187236
```

11.3. Generación de números enteros

Para generar un número entero aleatorio se puede usar la función `randint()`. A esta función se le deben entregar el número mínimo y el número máximo como parámetros.

Se puede usar para simular el lanzamiento de un dado.

```
import random

print(random.randint(1, 6))
```

Salida:

```
2
```

O lanzamientos de múltiples dados.

```
import random

for i in range(10):
    print(random.randint(1, 6))
```

Salida:

```
1
5
5
6
3
4
1
3
4
3
```

Alternativamente, se puede usar la función `randrange()`, que entrega el mismo resultado, pero usa los parámetros de un objeto de tipo `range`. Esto significa que el número máximo será el antecesor del segundo parámetro.

```
import random

for i in range(10):
    print(random.randrange(1, 6)) # entre 1 y 5
```

Salida:

```
4
1
5
5
3
1
2
1
2
1
```

11.4. Generación de números racionales

La función `random()` sirve para generar un float entre `0` y `1`. Este resultado se puede modificar con operaciones para obtener resultados dentro de un rango específico.

Pero puede resultar más fácil utilizar la función `uniform()`, que requiere 2 argumentos al igual que `randint`, pero genera un número de tipo float.

```
import random

print(random.uniform(-10, 10))
print(random.uniform(25.7, 75.3))
```

Salida:

```
-8.19726009116228
70.86548956595794
```

11.5. Semillas

Una semilla es un valor predefinido que se le entrega al algoritmo generador de números aleatorios.

Por defecto, la semilla es la hora actual del sistema en el que se ejecuta el algoritmo, pero se puede cambiar usando el método `seed()`.

```
import random

random.seed(10)
print(random.random())
```

Salida:

```
0.5714025946899135
```

El resultado obtenido en este caso siempre será el mismo, ya que se le entregó la semilla inicial `10`.

Sin importar cuantas veces se repita, el primer resultado siempre será igual para una misma semilla.

```
import random

for i in range(5):
    random.seed(10)
    print(random.random())
```

Salida:

```
0.5714025946899135
0.5714025946899135
0.5714025946899135
0.5714025946899135
0.5714025946899135
```

El resto de resultados también serán iguales, y se repetirán en el mismo orden.

```
import random

for i in range(2):
    random.seed(10)
    print(random.random())
    print(random.random())
    print(random.random())
    print()
```

Salida:

```
0.5714025946899135
0.4288890546751146
0.5780913011344704

0.5714025946899135
0.4288890546751146
0.5780913011344704
```

La misma semilla se usará con todos los métodos del módulo **random**.

Los argumentos que se le entregan a funciones como **randint()** no influyen en el resultado obtenido con la semilla, sólo lo ajustan al rango.

```
import random

# x1
random.seed(50)
print(random.randint(1, 10))
print(random.randint(1, 10))
print(random.randint(1, 10))
print(random.randint(1, 10))
print()

# x2
random.seed(50)
print(random.randint(1, 20))
print(random.randint(1, 20))
print(random.randint(1, 20))
print(random.randint(1, 20))
print()

# x4
random.seed(50)
print(random.randint(1, 40))
print(random.randint(1, 40))
print(random.randint(1, 40))
print(random.randint(1, 40))
```

Salida:

```
8
5
6
4

16
9
12
8

32
18
24
16
```

```
import random

# x1
random.seed(50)
print(random.randint(1, 10))
print(random.randint(1, 10))
print(random.randint(1, 10))
print(random.randint(1, 10))
print()

# +10
random.seed(50)
print(random.randint(11, 20))
print(random.randint(11, 20))
print(random.randint(11, 20))
print(random.randint(11, 20))
print()

# -11
random.seed(50)
print(random.randint(-10, -1))
print(random.randint(-10, -1))
print(random.randint(-10, -1))
print(random.randint(-10, -1))
```

Salida:

```
8
5
6
4

18
15
16
14

-3
-6
-5
-7
```

11.6. Generación de listas aleatorias

No existe una función para generar listas con números aleatorios, pero se puede realizar esta acción usando un bucle `for`.

```
import random

lista_enteros = []

for i in range(10):
    n = random.randint(1, 30)
    lista_enteros.append(n)

print(lista_enteros)
```

Salida:

```
[28, 29, 27, 1, 2, 17, 16, 8, 14, 1]
```

```
import random

lista_floats = []

for i in range(10):
    n = random.uniform(1, 30)
    lista_floats.append(n)

print(lista_floats)
```

Salida:

```
[20.318057476943455, 28.94636885611087, 28.55504745890174, 13.580765877089242,
↪ 28.13858047700271, 3.0084664484042736, 28.253140312391, 26.97455854190084,
↪ 26.875672112032685, 18.885133092791907]
```

Otra forma de generar listas de números enteros aleatorios es usando la función `sample()`.

```
import random

# genera 10 números aleatorios entre 10 y 30
lista_enteros = random.sample(range(10, 31), 10)

print(lista_enteros)
```

Salida:

```
[26, 13, 28, 12, 24, 25, 30, 16, 22, 15]
```

11.7. Elementos al azar de secuencias

El módulo `random` también tiene métodos para trabajar aleatoriamente con secuencias (como cadenas de texto y listas).

El método `choice()` retorna un elemento aleatorio de una secuencia.

```
import random

lenguajes = ["Python", "C", "C++", "Java", "C#", "Swift", "Ruby", "Kotlin"]

print(random.choice(lenguajes))
```

Salida:

```
Python
```

```
import random

python = "Python"

for i in range(10):
    print(random.choice(python))
```

Salida:

```
P
h
n
y
P
h
y
o
t
n
```

La función `sample()` obtiene una muestra aleatoria de una secuencia. Se le debe especificar el tamaño de la muestra en el segundo argumento. La secuencia original no cambia.

```
import random

alfabeto = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'ñ',
            'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z']

# muestra de 5 elementos
print(random.sample(alfabeto, 5))

# muestra de 10 elementos
print(random.sample(alfabeto, 10))

# muestra de 1 elemento
print(random.sample(alfabeto, 1))
```

Salida:

```
['l', 'w', 'e', 'a', 'p']
['v', 'p', 'y', 'o', 'n', 'c', 'q', 'ñ', 'm', 'w']
['f']
```

No se pueden tomar muestras de tamaño superior al tamaño de la población (secuencia) o de tamaño negativo.


```
import random

alfabeto = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'ñ',
            'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z']

print(random.sample(alfabeto, len(alfabeto) + 1))
```

Salida:

```
Traceback (most recent call last):
  File "<stdin>", line 5, in <module>
    print(random.sample(alfabeto, len(alfabeto) + 1))
  File "<stdin>", line 450, in sample
    raise ValueError("Sample larger than population or is negative")
ValueError: Sample larger than population or is negative
```

```
import random

alfabeto = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'ñ',
            'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z']

print(random.sample(alfabeto, -1))
```

Salida:

```
Traceback (most recent call last):
  File "<stdin>", line 5, in <module>
    print(random.sample(alfabeto, -1))
  File "<stdin>", line 450, in sample
    raise ValueError("Sample larger than population or is negative")
ValueError: Sample larger than population or is negative
```

El método **shuffle()** desordena una secuencia, y vuelve a armarla ordenándola al azar. Cambia la lista original, no retorna una nueva.

```
import random

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

random.shuffle(numeros)
print(numeros)

random.shuffle(numeros)
print(numeros)
```

Salida:

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

La función **choices()** permite seleccionar elementos de una lista al azar **k** veces. Esta acción se realiza “con reposición”, o sea, después de seleccionar un elemento este elemento seguirá en la lista. Este número puede ser mayor al número de elementos de la lista.

```
import random

frutas = ["manzana", "naranja", "frutilla"]

# 1 elección
print(random.choices(frutas))

# 10 elecciones
print(random.choices(frutas, k=10))
```

Salida:

```
['frutilla']
['manzana', 'naranja', 'frutilla', 'frutilla', 'naranja', 'manzana', 'frutilla',
↪ 'manzana', 'naranja', 'manzana']
```

Todos los elementos tendrán la misma probabilidad de ser escogidos.

Se le puede entregar el argumento **weights** para hacer que cada elemento tenga su propia probabilidad de ser escogido, por ejemplo un dado truco.

```
import random

dado_truco = [1, 2, 3, 4, 5, 6]

# dado truco para que sea más probable obtener un 3
# dado_truco = [1, 2, 3, 4, 5, 6], 10 tiradas
print(random.choices(dado_truco, weights=[1, 1, 6, 1, 1, 1], k=10))
```

Salida:

```
[3, 3, 5, 3, 3, 3, 3, 5, 2, 3]
```

Los números dentro de la lista **weights** representan el peso de cada elemento de la secuencia, y deben tener la misma medida.

Los números pueden escribirse en cualquier escala, ya que se comparan entre ellos para determinar la razón en la que se encuentran los pesos. Todos los ejemplos que se muestran a continuación son equivalentes.

```
import random

frutas = ["manzana", "naranja", "frutilla"]

# los siguientes ejemplos son equivalentes
random.seed(100)
print(random.choices(frutas, weights=[2, 6, 3], k=5))

random.seed(100)
print(random.choices(frutas, weights=[20, 60, 30], k=5))

random.seed(100)
print(random.choices(frutas, weights=[0.2, 0.6, 0.3], k=5))

random.seed(100)
print(random.choices(frutas, weights=[2000000, 6000000, 3000000], k=5))
```

Salida:

```
['manzana', 'naranja', 'frutilla', 'naranja', 'frutilla']
['manzana', 'naranja', 'frutilla', 'naranja', 'frutilla']
['manzana', 'naranja', 'frutilla', 'naranja', 'frutilla']
['manzana', 'naranja', 'frutilla', 'naranja', 'frutilla']
```

Por último, en vez de `weights` puede usarse `cum_weights`, para indicar los pesos acumulados.

```
import random

frutas = ["manzana", "naranja", "frutilla"]

# los siguientes ejemplos son equivalentes
random.seed(1000)
print(random.choices(frutas, weights=[2, 6, 3], k=5))

random.seed(1000)
print(random.choices(frutas, cum_weights=[2, 8, 11], k=5))
# [2, 2+6, 2+6+3]
```

Salida:

```
['frutilla', 'naranja', 'manzana', 'naranja', 'naranja']
['frutilla', 'naranja', 'manzana', 'naranja', 'naranja']
```

11.8. Distribuciones de probabilidad

El módulo `random` también contiene funciones para generar números al azar siguiendo ciertas distribuciones de probabilidad.

La función `uniform()` retorna un float que sigue la [distribución uniforme](#), que se encuentra entre 2 valores.

```
import random

print(random.uniform(1, 100))
```

La función `triangular()` retorna un float que sigue la [distribución triangular](#), que se encuentra entre 2 valores y se le puede entregar el parámetro `mode` (valor más probable). Si no se le entrega este argumento, `mode` será el punto medio.

```
import random

print(random.triangular(1, 100))

print(random.triangular(1, 100, 20))
```

El resto de distribuciones no serán detalladas, ya que su uso es más específico, pero se mostrará el nombre de cada una y el nombre de la función para generar floats al azar que la sigan.

- [Distribución beta](#): `betavariate()`
- [Distribución exponencial](#): `expovariate()`
- [Distribución gamma](#): `gammavariate()`
- [Distribución log-normal](#): `lognormvariate()`
- [Distribución normal](#): `normalvariate()` y `gauss()`

- Distribución de von Mises: `vonmisesvariate()`
- Distribución de Pareto: `paretovariate()`
- Distribución de Weibull: `weibullvariate()`

Se puede usar la función `help()` para buscar los parámetros de cada una.

Capítulo 12

Excepciones

12.1. Excepciones

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

Por ejemplo, al intentar dividir por 0, se produce un `ZeroDivisionError`.

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

Salida:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    print(10 / 0)
ZeroDivisionError: division by zero
```

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.

Una lista más detallada de excepciones se puede encontrar en la [documentación oficial](#).

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 ejecutará.

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

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

Entrada:

```
3
4
```

Salida:

```
Resultado de la división: 0.75
```

Entrada:

```
10
0
```

Salida:

```
Error: Se intentó dividir por 0
```

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

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

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

Entrada:

```
10
a
```

Salida:

```
Error: Valor o tipo no válido
```

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

```
palabra = 'spam'
numero = 0

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

Salida:

```
Ha ocurrido un error
```

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

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

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

12.3. Declaración finally

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

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

Salida:

```
Hola
Error: División por 0
Este mensaje siempre se mostrará
```

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

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

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

Salida:

```

1
Este mensaje siempre se mostrará
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
    print(1 / 0)
ZeroDivisionError: division by zero

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 5, in <module>
    print(variable_desconocida)
NameError: name 'variable_desconocida' is not defined

```

Si ocurre una excepción sin atrapar, la excepción se muestra después de lo que contenía la expresión **finally**.

12.4. Levantar excepciones

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

```

print(1)
raise ValueError
print(2)

```

Salida:

```

1
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    raise ValueError
ValueError

```

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

```

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

```

Salida:

```

Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    raise NameError("¡Nombre no válido!")
NameError: ¡Nombre no válido!

```

```

password = input("Ingresa tu contraseña: ")

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

```

Entrada:

```

hunter2

```


Salida:

```
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
    raise ValueError("La contraseña debe tener 8 o más caracteres")
ValueError: La contraseña debe tener 8 o más caracteres
```

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

```
try:
    print(7 / 0)
except:
    print("Ha ocurrido un error")
    raise # vuelve a levantar una excepción ya atrapada
```

Salida:

```
Ha ocurrido un error
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    print(7 / 0)
ZeroDivisionError: division by zero
```

Usar `raise` fuera de los contextos mencionados anteriormente levantará un `RuntimeError`.

```
raise
```

Salida:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    raise
RuntimeError: No active exception to reraise
```

12.5. Aserciones

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

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

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

Salida:

```
a
b
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
    assert 1 > 10 # False
AssertionError
```

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

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

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

Salida:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    assert 1 == 100, "Ha ocurrido un error"
AssertionError: Ha ocurrido un error
```

Las excepciones `AssertionError` pueden ser atrapadas y manejadas como cualquier otra excepción utilizando la sentencia `try-except`, pero si no son manejadas, terminarán la ejecución el programa, como cualquier otra excepción.

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.

Las declaraciones `assert` no pueden ir sin un predicado.

```
assert
```

Salida:

```
File "<stdin>", line 1
  assert
    ^
SyntaxError: invalid syntax
```

12.6. Validación de entrada

El ingreso de datos en cualquier programa es algo que se debe validar para que el programa trabaje correctamente.

Supongamos que se quiere que el usuario ingrese un número. Esto se haría usando la función `input()` para tomar entrada y la función `int()` para convertirla a un número entero.

Un problema que puede ocurrir es si el usuario ingresa una letra o símbolo que no puede ser convertido a un número entero.

```
num = int(input('Ingrese un número: '))
```

Entrada:

```
siete
```

Salida:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    num = int(input('Ingrese un número: '))
ValueError: invalid literal for int() with base 10: 'siete'
```

Entrada:

2.5

Salida:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    num = int(input('Ingrese un número: '))
ValueError: invalid literal for int() with base 10: '2.5'
```

Para manejar este error, se debería usar un **try-except** para atrapar la excepción **ValueError**.

```
try:
    num = int(input('Ingrese un número: '))
except ValueError:
    print('Debes escribir un número')
```

Entrada:

hola

Salida:

Debes escribir un número

Esto soluciona el problema que ocurre por la excepción **ValueError** no capturada, pero hace falta el uso de un bucle para pedir la entrada nuevamente en caso de que no sea válida.

```
while True:
    try:
        num = int(input('Ingrese un número: '))
    except ValueError:
        print('Debes escribir un número')
        continue

    break

print('Entrada válida')
```

Entrada:

abc
10

Salida:

Debes escribir un número
Entrada válida

Supongamos que el programa necesita pedir la edad del usuario en vez de un número cualquiera. Eso significa que los valores negativos no son válidos, lo cual puede solucionarse usando una declaración **if-else**.

```

while True:
    try:
        edad = int(input('Ingrese su edad: '))
    except ValueError:
        print('Debes escribir un número')
        continue

    if edad < 0:
        print('Debes escribir un número positivo')
        continue
    else:
        break

print('Entrada válida')

```

Entrada:

```

veinte
-20
20

```

Salida:

```

Debes escribir un número
Debes escribir un número positivo
Entrada válida

```

Esa es la estructura general para cualquier validación de entrada. Primero se atrapan las excepciones y después se evalúan los casos particulares no excepcionales que no se consideran válidos.

Lo más importante de la validación de entrada es asumir que el usuario puede hacer cualquier acción en cualquier momento. El programador debe anteponerse a todos los casos posibles y manejarlos para evitar cualquier error que pueda ocurrir durante el tiempo de ejecución.

Capítulo 13

Archivos

13.1. Abrir archivos

Python se puede usar para leer y escribir los contenidos de archivos. Los archivos de texto son los más fáciles de manipular.

Antes de que un archivo pueda ser editado, debe ser abierto con la función `open()`.

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

El argumento de la función `open()` es la ruta del archivo. Esta ruta puede ser relativa (desde el directorio de trabajo) o absoluta.

Si se necesita, se puede hacer un procedimiento un poco más largo para abrir un archivo relativamente desde el directorio donde se encuentra el archivo `.py`, sin importar cual sea el directorio de trabajo o sistema operativo.

```
import os

directorio_script = os.path.dirname(os.path.abspath(__file__))
ruta_relativa = "archivo.txt"
ruta_absoluta = os.path.join(directorio_script, ruta_relativa)

archivo = open(ruta_absoluta)

# para ver las rutas y entender qué ocurrió
print(directorio_script)
print(ruta_relativa)
print(ruta_absoluta)
```

En este caso, `ruta_relativa` es lo único que debe cambiarse para abrir un archivo distinto dentro del mismo directorio.

13.2. Modos de apertura

Se puede especificar el modo utilizado para abrir un archivo al pasar un segundo argumento a la función `open()`. Los modos de apertura son:

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

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

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

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

- **"x "**: Significa modo de creación exclusiva, se usa para crear un fichero y escribir sobre él.

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

Entrega un error **FileExistsError** si el archivo ya existe y no está vacío.

```
# Modo de creación, pero el archivo ya existe y tiene contenido
archivo = open("archivo.txt", "x")
```

Archivo inicial:

hola

Salida:

```
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    archivo = open("archivo.txt", "x")
FileExistsError: [Errno 17] File exists: 'archivo.txt'
```

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

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

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

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

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

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

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

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

13.3. Extensión de modos de apertura

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

La siguiente tabla muestra el funcionamiento de cada modo:

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

*: Levanta una excepción `FileExistsError` si el archivo ya tiene contenido.

13.4. Cierre de archivos

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

```
archivo = open("archivo.txt", "w")  
  
# Trabajar con el archivo...  
  
archivo.close()
```

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

13.5. Lectura de archivos

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

```
archivo = open("archivo.txt", "r")  
  
contenidos = archivo.read()  
archivo.close() # Ya se almacenaron los contenidos del archivo en una variable  
  
print(contenidos)
```

Archivo inicial:

```
hola, soy un archivo
```

Salida:

```
hola, soy un archivo
```

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

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

```

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

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

archivo.close()

```

Archivo inicial:

```

hola, soy un archivo, ahora lees el texto que almaceno

```

Salida:

```

hola, soy un arc
hivo, ah
ora
lees el texto que almaceno

```

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

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

```

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

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

archivo.close()

```

Archivo inicial:

```

hola, soy un archivo, ahora lees el texto que almaceno

```

Salida:

```

hola, soy un archivo, ahora lees el texto que almaceno

```

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

```

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

print(archivo.read())
archivo.seek(0) # Regresa a la posición 0
print(archivo.read())

archivo.close()

```

Archivo inicial:

```

hola, soy un archivo, ahora lees el texto que almaceno

```

Salida:


```
hola, soy un archivo, ahora lees el texto que almaceno
hola, soy un archivo, ahora lees el texto que almaceno
```

Alternativamente, se le puede decir a `seek()` que se mueva a otra posición.

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

print(archivo.read())
archivo.seek(16) # Regresa a la posición 16
print(archivo.read())

archivo.close()
```

Archivo inicial:

```
hola, soy un archivo, ahora lees el texto que almaceno
```

Salida:

```
hola, soy un archivo, ahora lees el texto que almaceno
hivo, ahora lees el texto que almaceno
```

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

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

print(archivo.read(16))
print(archivo.tell())

archivo.close()
```

Archivo inicial:

```
hola, soy un archivo, ahora lees el texto que almaceno
```

Salida:

```
hola, soy un arc
16
```

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.

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

print(archivo.readlines())

archivo.close()
```

Archivo inicial:

```
Roses are Red,
Violets are Blue,
Unexpected '{'
on line 32.
```

Salida:

```
['Roses are Red,\n', 'Violets are Blue,\n', "Unexpected '{'\n", 'on line 32.']
```

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

```
archivo = open("archivo.txt", "r")  
  
for linea in archivo:  
    print(linea)  
  
archivo.close()
```

Archivo inicial:

```
Roses are Red,  
Violets are Blue,  
Unexpected '{'  
on line 32.
```

Salida:

```
Roses are Red,  
  
Violets are Blue,  
  
Unexpected '{'  
  
on line 32.
```

13.6. Escritura de archivos

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

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

Archivo inicial:

Archivo final:

```
Hola mundo
```

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

```
# Se borran los contenidos del archivo y se escribe sobre ellos  
archivo = open("archivo.txt", "w")  
archivo.write("Archivo final")  
archivo.close()
```

Archivo inicial:

Archivo inicial

Archivo final:

Archivo final

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

```
archivo = open("archivo.txt", "a")
archivo.write("hola")
archivo.close()
```

Archivo inicial:

...

Archivo final:

...hola

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

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

bytes_escritos = archivo.write("Lorem ipsum dolor sit amet, consectetur adipiscing elit,
→ sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim
→ veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo
→ consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum
→ dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident,
→ sunt in culpa qui officia deserunt mollit anim id est laborum.")
print(bytes_escritos)

archivo.close()
```

Archivo inicial:

Archivo final:

```
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
→ incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud
→ exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute
→ irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla
→ pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui
→ officia deserunt mollit anim id est laborum.
```

Salida:

430

Para escribir algo que no sea un string, necesita ser convertido primero usando el método `str()`.

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

archivo.write(str(42))

archivo.close()
```

Archivo inicial:

Archivo final:

13.7. Declaración with

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

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

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

Al usar declaraciones `with`, se crea una variable temporal (a menudo llamada `f`), la cual solo es accesible en el bloque indentado de la declaración `with`.

```
with open("archivo.txt") as f:
    # Trabajar con el archivo...
```

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

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

```
try:
    with open("archivo.txt") as f:
        # Trabajar con el archivo...
except:
    print("Error")
```

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

Si se necesita mantener un archivo abierto por mucho tiempo, no se recomienda usar `with`.

Capítulo 14

Módulos time y datetime

Capítulo 15

Estructuras de datos

15.1. Estructuras de datos

15.2. Diccionarios

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

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

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

Salida:

```
25
42
```

Cada elemento de un diccionario es representado por un par clave:valor (key:value). Los elementos dentro de diccionarios no están ordenados, lo que significa que no pueden ser indexados por índices, sólo por claves.

Un diccionario puede almacenar como valor cualquier tipo de datos. Se pueden usar saltos de línea para definirlos de forma más ordenada.

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

print(primarios["verde"])
```

Salida:

```
[0, 255, 0]
```

15.3. Listas como diccionarios

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

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

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

Salida:

```
Violín
Piano
Guitarra
Batería
```

15.4. Diccionario vacío

Un diccionario vacío es definido como `{}`.

```
diccionario_vacio = {}
```

15.5. Excepciones en diccionarios

Tratar de indexar una clave que no es parte de un diccionario retorna un `KeyError`.

```
idiomas = {
    "es": "Español",
    "en": "Inglés",
    "it": "Italiano",
    "ja": "Japonés"
}

print(idiomas["fr"])
```

Salida:

```
Traceback (most recent call last):
  File "<stdin>", line 8, in <module>
    print(idiomas["fr"])
KeyError: 'fr'
```

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`.

```
dic = {[1, 2, 3]: "uno, dos, tres"}
```

Salida:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    dic = {[1, 2, 3]: "uno, dos, tres"}
TypeError: unhashable type: 'list'
```

15.6. Indexación de diccionarios

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

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

cuadrados[6] = 36
cuadrados[3] = 9
print(cuadrados)
```

Salida:

```
{1: 1, 2: 4, 3: 9, 4: 16, 6: 36}
```

15.7. Uso de in y not en diccionarios

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

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

print(1 in numeros)
print("tres" in numeros)
print(4 not in numeros)
```

Salida:

```
True
False
True
```

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

15.8. Función get()

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

```
pares = {
    1: "manzana",
    "naranja": [2, 3, 4],
    True: False,
    None: "True"
}

print(pares.get("naranja"))
print(pares.get(42))
print(pares.get(12345, "no encontrado"))
```

Salida:


```
[2, 3, 4]
None
no encontrado
```

En este caso es importante recordar que **1** y **True** significan lo mismo, y por lo tanto son la misma clave. Entonces, el diccionario **pares** sólo tiene 3 elementos (la clave **True** tendrá como valor **False**, el último valor que se le asignó).

```
pares = {
    1: "manzana",
    "naranja": [2, 3, 4],
    True: False,
    None: "True"
}

print(pares)
```

Salida:

```
{1: False, 'naranja': [2, 3, 4], None: 'True'}
```

15.9. Convertir diccionarios a listas

Usando el método **list()**, se puede convertir un diccionario a una lista, pero esto sólo obtendrá sus claves.

```
diccionario = {"a": 1, "b": 2, "c": 3}
lista = list(diccionario)

print(lista)
```

Salida:

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

Los métodos **keys()** y **values()** permiten obtener las claves y los valores de un diccionario.

```
diccionario = {"a": 1, "b": 2, "c": 3}

print(diccionario.keys())
print(diccionario.values())
```

Salida:

```
dict_keys(['a', 'b', 'c'])
dict_values([1, 2, 3])
```

El resultado de estos métodos puede ser convertido a listas usando el método **list()** para poder manipularlos con más facilidad.

```

diccionario = {"a": 1, "b": 2, "c": 3}

claves = list(diccionario.keys())
valores = list(diccionario.values())

print(claves)
print(valores)

```

Salida:

```

['a', 'b', 'c']
[1, 2, 3]

```

15.10. Tuplas

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

```

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

```

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

15.11. Indexación de Tuplas

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

```

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

print(vehiculos[2])
print(vehiculos[4])

```

Salida:

```

Avión
Tren

```

Los datos dentro de una tupla están ordenados, empezando desde el índice 0.

15.12. Excepciones en tuplas

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

```

animales = ("mono", "elefante")

animales[1] = "gato"

```

Salida:

```

Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
    animales[1] = "gato" # TypeError
TypeError: 'tuple' object does not support item assignment

```

15.13. Anidación de tuplas

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.

```
tupla = (1, "hola", ['a', 'b'], (10, 20))
tupla[2][0] = 'x'

print(tupla)
```

Salida:

```
(1, 'hola', ['x', 'b'], (10, 20))
```

15.14. Tupla vacía

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

```
tupla_vacia = ()
```

15.15. Manejo de variables con tuplas

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

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

print(f1)
print(f2)
print(f3)
```

Salida:

```
manzana
naranja
frutilla
```

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

```
numeros = "uno", "dos", "tres"
print(numeros)
```

Salida:

```
('uno', 'dos', 'tres')
```

Se pueden usar tuplas para intercambiar los valores de 2 variables, sin necesidad de crear una variable auxiliar.

```
siete = 20
veinte = 7

# truco con tuplas
siete, veinte = veinte, siete

# se intercambiaron los valores
print(siete)
print(veinte)
```

Salida:

```
7
20
```

15.16. 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` y `not in` para revisar si contienen o no un elemento en particular.

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

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

Salida:

```
True
False
```

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

```
lista = ['a', 'b', 'c', 'c', 'a', 'd', 'b', 'e', 'd', 'c', 'b']
conjunto = set(lista)

print(lista)
print(conjunto) # sin elementos duplicados, sin orden
```

Salida:

```
['a', 'b', 'c', 'c', 'a', 'd', 'b', 'e', 'd', 'c', 'b']
{'d', 'a', 'e', 'c', 'b'}
```

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

15.17. Conjunto vacío

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

```
conjunto_vacio = set()
```

15.18. Métodos de conjuntos

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

```
felinos = {"gato", "león", "tigre"}  
  
print(len(felinos))
```

Salida:

```
3
```

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

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

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

Salida:

```
{1, 2, 3, 4, 5, 6}  
{1, 2, 4, 5, 6, -7}
```

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

Generalmente, elimina el primer elemento, pero esto no se puede garantizar para todos los casos.

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

15.19. Operaciones con conjuntos

Los conjuntos pueden ser combinados utilizando operaciones matemáticas.

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

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

Salida:

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

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

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

# Intersección
print(primero & segundo)
```

Salida:

```
{4, 5, 6}
```

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

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

# Diferencia
print(primero - segundo)
print(segundo - primero)
```

Salida:

```
{1, 2, 3}
{8, 9, 7}
```

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

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

# Diferencia simétrica
print(primero ^ segundo)

# Se obtiene el mismo resultado usando otras operaciones
print((primero | segundo) - (primero & segundo))
```

Salida:

```
{1, 2, 3, 7, 8, 9}
{1, 2, 3, 7, 8, 9}
```

15.20. Frozenset

Los frozenset son, como dice su nombre, conjuntos “congelados”. Se comportan exactamente igual que un conjunto normal, pero son inmutables. Para crearlos, se debe usar la función `frozenset()`.

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

numeros = frozenset(numeros)

print(numeros)
```

Salida:

```
frozenset({0, 1, 2, 3, 4, 5, 6, 7, 8, 9})
```

Como son inmutables, intentar cambiarlos causará una excepción. De hecho, ni siquiera existen métodos para modificarlos.

```
vocales = frozenset(['a', 'e', 'i', 'o'])  
vocales[0] = 'u'
```

Salida:

```
Traceback (most recent call last):  
  File "<stdin>", line 3, in <module>  
    vocales[0] = 'u'  
TypeError: 'frozenset' object does not support item assignment
```

```
vocales = frozenset(['a', 'e', 'i', 'o'])  
vocales.add('u')
```

Salida:

```
Traceback (most recent call last):  
  File "<stdin>", line 3, in <module>  
    vocales.add('u')  
AttributeError: 'frozenset' object has no attribute 'add'
```

```
vocales = frozenset(['a', 'e', 'i', 'o'])  
vocales.remove('a')
```

Salida:

```
Traceback (most recent call last):  
  File "<stdin>", line 3, in <module>  
    vocales.remove('a')  
AttributeError: 'frozenset' object has no attribute 'remove'
```

```
vocales = frozenset(['a', 'e', 'i', 'o'])  
vocales.append('u')
```

Salida:

```
Traceback (most recent call last):  
  File "<stdin>", line 3, in <module>  
    vocales.append('u')  
AttributeError: 'frozenset' object has no attribute 'append'
```

15.21. Resumen sobre estructuras de datos

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

¿Cuándo utilizar diccionarios?

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

¿Cuándo utilizar listas?

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

¿Cuándo utilizar conjuntos?

- Cuando se necesita que los elementos sean únicos.

¿Cuándo utilizar tuplas?

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

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

La siguiente tabla muestra las propiedades de cada estructura de datos vista hasta ahora. También se muestran cadenas ya que tienen propiedades similares.

| | Cadena | Lista | Diccionario | Tupla | Conjunto | Frozenset |
|------------------|------------|---------|-------------|---------|-------------|-------------|
| Mutable | No | Sí | Sí | No | Sí | No |
| Indexado | Índices | Índices | Claves | Índices | No | No |
| Secuencial | Sí | Sí | No | Sí | No | No |
| Elementos únicos | No | No | No | No | Sí | Sí |
| Tipos de datos | Caracteres | Todos | Todos | Todos | Todos | Todos |
| Notación | ' ' o "" | [] | {:} | () | { } o set() | frozenset() |

Existen estructuras de datos más complejas, pero no se verán todavía.

Capítulo 16

Iterables

16.1. Cortes de lista

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

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

print(cuadrados[2:6])
print(cuadrados[3:8])
print(cuadrados[0:1])
```

Salida:

```
[4, 9, 16, 25]
[9, 16, 25, 36, 49]
[0]
```

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

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

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

print(cuadrados[:5])
print(cuadrados[5:])
print(cuadrados[:])
```

Salida:

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

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

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

print(cuadrados[::2])
print(cuadrados[2:8:3])
```

Salida:

```
[0, 4, 16, 36, 64]
[4, 25]
```

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

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

Salida:

```
[1, 4, 9, 16, 25, 36, 49, 64]
```

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

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

Salida:

```
[4, 3, 2, 1]
```

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

Se pueden usar cortes para copiar listas, aunque no suele ser la forma más clara de hacerlo.

```
a = ['x', 'y']
b = a[:]
b[0] = 'z'

print(b)
print(a)
```

Salida:

```
['z', 'y']
['x', 'y']
```

16.2. Cortes de tuplas

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

```
tupla = (1, 2, 3, 4, 5, 6, 7, 8)
print(tupla[2:5]) # (3, 4, 5)
```

Salida:

```
(3, 4, 5)
```

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

16.3. Listas por compresión

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

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

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

Salida:

```
[0, 1, 8, 27, 64]
```

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.

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

print(cuadrados_pares)
```

Salida:

```
[0, 4, 16, 36, 64]
```

Intentar crear una lista de rango demasiado extenso resultará en un **MemoryError**. Es posible que la operación tarde unos minutos en llegar a esta excepción, o que el PC se quede sin memoria antes de que llegue a aparecer.

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

Se debe recordar que se puede interrumpir la ejecución del programa en cualquier momento enviando un **KeyboardInterrupt**, al pulsar **Ctrl+C**.

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

16.4. 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.

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

print(mensaje)
```

Salida:

```
Números: 4, 5, 6
```

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

Los índices entre llaves `{}` pueden repetirse cuantas veces se quiera.

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

Salida:

```
abracadabra
```

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

```
mensaje = "Números: {0}, {1}, {3}".format(4, 5, 6, 7) # 6 (posición 2) nunca es usado
print(mensaje)
```

Salida:

```
Números: 4, 5, 7
```

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

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

Salida:

```
Números: 10, 11, 12
```

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

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

Salida:

```
5, 12
```

16.5. Funciones de cadenas

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

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

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

Salida:

```
rojo | azul | amarillo
```

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

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

Salida:

```
Hola mundo
```

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

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

Salida:

```
True  
True
```

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

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

Salida:

```
True  
True
```

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

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

```
texto = 'Python es un lenguaje de programación'  
  
print('lenguaje' in texto)  
print('idioma' in texto)  
  
# Al revés  
print('idioma' not in texto)  
print('lenguaje' not in texto)
```

Salida:

```
True
False
True
False
```

- **upper()**: Cambia una cadena a mayúsculas.

```
frase = "Esta es una frase."
print(frase.upper())
```

Salida:

```
ESTA ES UNA FRASE.
```

- **lower()**: Cambia una cadena a minúsculas.

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

Salida:

```
esta es una frase.
```

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

```
frase = "ESTA ES UNA FRASE."
print(frase.capitalize())
```

Salida:

```
Esta es una frase.
```

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

```
frase = "ESTA ES UNA FRASE."
print(frase.title())
```

Salida:

```
Esta Es Una Frase.
```

Al método **split()** visto anteriormente 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...']

```

16.6. Funciones all() y any()

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

```

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

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

```

Salida:

```
Todos los números son mayores que 5
```

```

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

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

```

Salida:

```
Al menos un número es par
```

16.7. Función enumerate()

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

```

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

for v in enumerate(nums):
    print(v)

```

Salida:

```

(0, 55)
(1, 44)
(2, 33)
(3, 22)
(4, 11)

```

También se puede especificar el índice desde el que comienza.

```
frutas = ["manzana", "naranja", "pera", "limón", "durazno", "uva", "cereza"]  
  
for f in enumerate(frutas, -3):  
    print(f)
```

Salida:

```
(-3, 'manzana')  
(-2, 'naranja')  
(-1, 'pera')  
(0, 'limón')  
(1, 'durazno')  
(2, 'uva')  
(3, 'cereza')
```


Capítulo 17

Programación funcional

17.1. Paradigma de programación funcional

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

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

```
def repetir_2_veces(funcion, arg):  
    return funcion(funcion(arg))  
  
def sumar5(x):  
    return x + 5  
  
print(repetir_2_veces(sumar5, 10))
```

Salida:

```
20
```

La función `repetir_2_veces()` mostrada en el ejemplo recibe otra función como su argumento y la llama 2 veces dentro de su cuerpo.

Entender los conceptos de programación funcional requiere un conocimiento básico del concepto algebraico llamado [“composición de funciones”](#).

17.2. 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 el mismo resultado para el mismo valor de x .

Ejemplo de función pura:

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

Ejemplo de función impura:

```
lista = []  
  
def impura(arg):  
    lista.append(arg)
```

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

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

Utilizar funciones puras tiene sus ventajas y desventajas.

Las funciones puras son:

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

Desventajas principales:

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

17.3. Lambdas

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

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

Lo mismo es posible con las funciones, dado que sean creadas utilizando la sintaxis `lambda`. Funciones creadas de esta forma son conocidas como anónimas.

Este enfoque es más comúnmente utilizado cuando se pasa una función sencilla como argumento de otra función.

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

```
def func(f, arg):  
    return f(arg)  
  
y = func(lambda x: 2*x*x, 5)  
print(y)
```

Salida:

50

Las funciones `lambda` reciben su nombre del [cálculo lambda](#), el cual es un modelo computacional inventado por Alonzo Church.

Las funciones `lambda` no son tan potentes como las funciones con nombre. Sólo pueden hacer cosas que requieren de una sola expresión, normalmente equivalente a una sola línea de código.

```
# función con nombre
def polinomio(x):
    return x**2 + 5*x - 4
print(polinomio(3))

# función lambda
print((lambda x: x**2 + 5*x - 4)(3))
```

Salida:

```
20
20
```

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

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

Salida:

```
14
```

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

17.4. Función map()

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

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

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

nums = [11, 22, 33, 44, 55]
resultado = list(map(sumar5, nums))
print(resultado)
```

Salida:

```
[16, 27, 38, 49, 60]
```

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

```
nums = [11, 22, 33, 44, 55]
resultado = list(map(lambda x: x + 5, nums))

print(resultado)
```

Salida:

```
[16, 27, 38, 49, 60]
```

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

17.5. 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).

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

print(resultado)
```

Salida:

```
[22, 44]
```

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

17.6. Generadores

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

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

Pueden ser creados utilizando funciones y la sentencia `yield`.

```
def cuenta_regresiva():
    i = 5
    while i > 0: # condición de salida
        yield i
        i -= 1

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

Salida:

```
5
4
3
2
1
```

Nótese que si en el ejemplo anterior no se hubiera especificado la condición de salida, hubiera seguido contando infinitamente en los negativos y tendría que detenerse pulsando `Ctrl+C`.

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

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

```
def infinitos_sietes():
    while True:
        yield 7

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

Salida:

```
7
7
7
7
7
7
7
7
.
.
.
```

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

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

```
from math import sqrt

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

    return True

def generar_primos():
    n = 2

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

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

Salida:

```
2
3
5
7
11
13
17
.
.
.
```

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

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

print(list(numeros(11)))
```

Salida:

```
[0, 2, 4, 6, 8, 10]
```

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

17.7. Decoradores

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

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

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

texto_decorado = decor(imprimir_texto)
texto_decorado()
```

Salida:

```
=====
¡Hola mundo!
=====
```

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

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

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

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

imprimir_texto()
print()

imprimir_texto = decor(imprimir_texto)
imprimir_texto()
print()

imprimir_texto = decor(imprimir_texto)
imprimir_texto()
```

Salida:

```
¡Hola mundo!

=====
¡Hola mundo!
=====

=====
=====
¡Hola mundo!
=====
=====
```

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

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

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

imprimir_texto = decor(imprimir_texto)
```

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

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

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

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

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

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

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

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

imprimir_texto()
```

Salida:

```
*****
=====
*****
¡Hola mundo!
*****
=====
*****
```

Los decoradores pueden ser usados para muchos otros propósitos además de “decorar”. El siguiente ejemplo muestra un decorador que añade la petición de entrada a funciones que no la tienen.


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

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

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

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

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

Un ejemplo aún más útil es un decorador para medir el tiempo que tarda en ejecutarse un algoritmo.

```
import time

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

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

suma_grande()
```

Salida:

```
Suma: 499999500000
Tiempo transcurrido: 283.50043296813965 ms
```

17.8. Recursión

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

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

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

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

print(factorial(5)) # 5 * 4 * 3 * 2 * 1
print(factorial(0)) # 1
print(factorial(-10))
```

Salida:

```
120
1
Traceback (most recent call last):
  File "<stdin>", line 11, in <module>
    print(factorial(-10))
  File "<stdin>", line 3, in factorial
    raise ValueError
ValueError
```

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

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

Abajo se muestra una versión incorrecta de la función factorial. No tiene caso base, así que se ejecutaría hasta que al interpretador se le acabe la memoria o se cuelgue, pero por suerte, el intérprete de Python es inteligente y levantará un `RecursionError`.

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

print(factorial(5))
```

Salida:

```
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
    print(factorial(5))
  File "<stdin>", line 2, in factorial
    return n * factorial(n-1)
  File "<stdin>", line 2, in factorial
    return n * factorial(n-1)
  File "<stdin>", line 2, in factorial
    return n * factorial(n-1)
  [Previous line repeated 996 more times]
RecursionError: maximum recursion depth exceeded
```

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

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

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

print(es_impar(17))
print(es_par(23))
```

Salida:

```
True
False
```

Otro ejemplo clásico es la serie de Fibonacci.

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

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

Salida:

```
0
1
1
2
3
5
8
13
21
34
```

17.9. Iteración vs. Recursión

17.10. Generadores recursivos

Capítulo 18

El módulo itertools

18.1. El módulo itertools

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

```
import itertools
```

18.2. Iteradores infinitos

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

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

```
from itertools import count

for i in count(3): # Empieza desde 3
    print(i) # Nunca termina
```

Salida:

```
3
4
5
6
7
8
9
10
.
.
.
```

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

```
from itertools import count

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

Salida:

```
3
5
7
9
11
13
15
17
19
21
```

Si el segundo parámetro es negativo, cuenta hacia atrás.

```
from itertools import count

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

Salida:

```
100
95
90
85
80
75
70
65
60
55
50
45
40
35
30
25
20
```

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

```
from itertools import cycle

for i in cycle("hola"):
    print(i)
```

Salida:

```
h  
o  
l  
a  
h  
o  
l  
a  
h  
.  
.  
.
```

```
from itertools import cycle  
  
for i in cycle([11, "hola", 33.5, False]):  
    print(i)
```

Salida:

```
11  
hola  
33.5  
False  
11  
hola  
33.5  
False  
11  
.  
.  
.
```

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

```
from itertools import repeat  
  
for i in repeat("hola", 3):  
    print(i)
```

Salida:

```
hola  
hola  
hola
```

Puede usarse para crear listas que tengan un objeto repetido.

```
from itertools import repeat  
  
holas = list(repeat("hola", 3))  
print(holas)
```

Salida:

```
['hola', 'hola', 'hola']
```

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

```
from itertools import repeat

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

Salida:

```
hola
hola
hola
.
.
.
```

18.3. Operaciones sobre iterables

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

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

```
from itertools import accumulate

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

print(list(range(8)))
print(nums)
```

Salida:

```
[0, 1, 2, 3, 4, 5, 6, 7]
[0, 1, 3, 6, 10, 15, 21, 28]
```

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

```
from itertools import takewhile

nums = range(10)

print(list(nums))
print(list(takewhile(lambda x: x <= 5, nums)))
```

Salida:

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

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

```
from itertools import takewhile

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

Salida:

```
[1, 2, 3]
[1, 2, 3, 3, 2, 1]
```

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

```
from itertools import chain

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

print(list(chain(lista1, lista2)))
```

Salida:

```
[1, 2, 3, 4, 5, 6]
```

18.4. Funciones de combinatoria

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

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

```
from itertools import product

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

print(list(product(letras1, letras2)))
```

Salida:

```
[('A', 'C'), ('A', 'D'), ('B', 'C'), ('B', 'D')]
```

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

```
from itertools import permutations

letras = ("A", "B")

print(list(permutations(letras)))
```

Salida:

```
[('A', 'B'), ('B', 'A')]
```



```
from itertools import permutations

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

print(list(permutations(letras)))
```

Salida:

```
[('A', 'B', 'C'), ('A', 'C', 'B'), ('B', 'A', 'C'), ('B', 'C', 'A'), ('C', 'A', 'B'),
↪ ('C', 'B', 'A')]
```

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

```
from itertools import permutations

letras = ("A", "B")

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

Salida:

```
[('A',), ('B',)]
[('A', 'B'), ('B', 'A')]
[]
```

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

```
from itertools import combinations

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

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

Salida:

```
[('A',), ('B',), ('C',)]
[('A', 'B'), ('A', 'C'), ('B', 'C')]
[('A', 'B', 'C')]
[]
```

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

Capítulo 19

Programación orientada a objetos

19.1. 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.

19.2. Objetos

Un objeto es una abstracción de un objeto en la vida real. Este objeto tiene características que lo describen (atributos) y comportamientos o acciones que puede realizar (métodos).

Ejemplo de un objeto:

- Persona:
- Posibles atributos: su nombre, su edad, su sexo, su altura, su peso, sus colores de pelo o piel, su país de residencia, etc.
- Posibles métodos: hablar, comer, dormir, caminar, correr, trabajar, pensar, etc.

Otra forma de verlo es que los objetos son sustantivos, los atributos son adjetivos y los métodos son verbos.

En términos más técnicos, un objeto es una instancia de una clase, y una clase es un molde usado para construir objetos. Con una misma clase, se pueden construir muchos objetos que comparten las mismas características, pero que tienen sus propios valores únicos para cada objeto.

19.3. Clases

Una 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.

Los objetos son una instancia de una clase, es decir, las clases dicen qué tiene un objeto, pero no indican los valores específicos de lo que tiene. Solo describen, no detallan.

Las clases son creadas utilizando la palabra clave `class` y un bloque indentado que contiene los métodos de una clase (los cuales son funciones) y puede contener sus atributos (que son variables) propias de la clase y sus objetos.

Para instanciar una clase y construir un objeto, sólo basta con llamar su método constructor usando el nombre de la clase y entregarle sus atributos como argumentos.

```
class Gato:
    # método __init__()
    def __init__(self, color):
        # atributo color
        self.color = color

# creación de clases, dándoles el atributo "color"
felix = Gato("negro")
tigre = Gato("café")
```

El código de arriba define una clase llamada **Gato**, la cual tiene el atributo **color**. Luego, la clase es utilizada para crear 2 objetos independientes de esa clase, los cuales son el **Gato felix** y el **Gato tigre**.

Las partes de una clase se verán en más detalle a continuación.

19.4. Constructor

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

Si el método `__init__()` va vacío, es decir, no ocurre nada al construir un objeto, no es necesario escribirlo.

```
class A:
    print("clase vacía")

a = A()
```

Salida:

```
clase vacía
```

```
class A:
    def __init__(self):
        print("clase vacía")

a = A()
```

Salida:

```
clase vacía
```

Ambos ejemplos mostrados arriba tienen el mismo efecto. Pero claramente, no ocurre nada interesante al crear objetos tan simples.

Veamos un ejemplo más complejo.

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

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

Todos los constructores de una clase deben tener **self** como su primer parámetro. No se necesita entregar **self** al momento de instanciar.

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

19.5. Atributos y self

Las instancias de una clase tienen atributos, los cuales son datos asociados a ellas, que definen sus características. En este ejemplo, las instancias de **Persona** tienen los atributos **nombre** y **edad**.

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

En el constructor `__init__()`, se le dan los parámetros **self**, **nombre** y **edad**. El parámetro **self** se refiere a la instancia, mientras que **nombre** y **edad** son los atributos de **Persona**.

El uso de **self** es para distinguir los atributos que se le entregan al constructor de los que tendrá la instancia. En este caso serán el mismo, pero se les puede dar valores por defecto.

```
class Persona:
    def __init__(self, nombre):
        self.nombre = nombre
        self.edad = 18 # todas las personas tendrán 18 años
```

Después, un objeto se construye dándole valores a sus atributos. En este ejemplo, tiene sentido pensar que **nombre** es un string y **edad** es un número.

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

# Juan es una persona de 25 años
juan = Persona("Juan", 25)
# María es una persona de 27 años
maria = Persona("María", 27)
```

Los atributos pueden ser accedidos al poner un punto seguido del nombre del atributo luego del nombre de una instancia.

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

juan = Persona("Juan", 25)
maria = Persona("María", 27)

print(juan.nombre)
print(maria.edad)
```

Salida:

```
Juan
27
```

El valor que tiene un atributo también puede ser cambiado durante la ejecución del programa.

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

a = Cuadrado("rojo")
print(a.color)

a.color = "verde"
print(a.color)
```

Salida:

```
rojo
verde
```

19.6. Métodos

Las clases pueden tener otros métodos definidos para agregarles funcionalidad. Todos los métodos deben tener **self** como su primer parámetro.

Estos métodos son accedidos utilizando la misma sintaxis de punto que los atributos.

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

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

# instanciando la clase
fido = Perro("Fido", "blanco")

# mostrando atributos
print(fido.nombre)
print(fido.color)
# llamando métodos
fido.ladrar()
```

Salida:

```
Fido
blanco
¡Guau!
```

19.7. Atributos de clase

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

```

class Perro:
    # atributo de clase
    patas = 4

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

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

# accedido desde un objeto
print(fido.patas)
# accedido desde la clase
print(Perro.patas)

```

Salida:

```

4
4

```

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

```

class Perro:
    patas = 4

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

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

# valores por defecto
print(fido.patas)
print(Perro.patas)

Perro.patas = 5
# cambian el valor para la clase y todos los objetos
print(fido.patas)
print(Perro.patas)

fido.patas = 6
# sólo cambia el valor para un objeto
print(fido.patas)
print(Perro.patas)

```

Salida:

```

4
4
5
5
6
5

```

19.8. Excepciones de clases

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

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

rect = Rectangulo(5, 6)
print(Rectangulo.color)
```

Salida:

```
Traceback (most recent call last):
  File "<stdin>", line 7, in <module>
    print(Rectangulo.color)
AttributeError: type object 'Rectangulo' has no attribute 'color'
```

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

rect = Rectangulo(5, 6)
Rectangulo.pintar("azul")
```

Salida:

```
Traceback (most recent call last):
  File "<stdin>", line 7, in <module>
    Rectangulo.pintar("azul")
AttributeError: type object 'Rectangulo' has no attribute 'pintar'
```

19.9. Herencia

La herencia brinda una forma de compartir funcionalidades entre clases.

Por ejemplo, las clases `Perro`, `Gato`, `Conejo`, etc. son parecidas. 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 entre ellas.

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

```

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

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

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

firulais = Perro("Firulais", "café")

print(firulais.nombre)
print(firulais.color)
firulais.ladrar()

print() # salto de línea, para separar

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

print(misifu.nombre)
print(misifu.color)
misifu.maullar()

```

Salida:

```

Firulais
café
¡Guau!

Misifu
blanco
¡Miau!

```

Una clase que hereda de otra clase se llama subclase. Una clase de la cual se hereda se llama superclase. Los métodos y atributos de la superclase son heredados por sus subclases.

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

```

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

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

class Perro(Lobo):
    # hereda el constructor de lobo

    # sobrescribe el método ladrar() de Lobo
    def ladrar(self):
        print("Guau")

husky = Perro("Max", "gris")
husky.ladrar()

```


Salida:

```
Guau
```

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

```
class A:
    def metodo1(self):
        print("Método de A")

class B(A):
    def metodo2(self):
        print("Método de B")

class C(B):
    def metodo3(self):
        print("Método de C")

c = C()

c.metodo1() # A
c.metodo2() # B
c.metodo3() # C
```

Salida:

```
Método de A
Método de B
Método de C
```

Sin embargo, no es posible la herencia circular.

```
class A(C):
    def metodo1(self):
        print("Método de A")

class B(A):
    def metodo2(self):
        print("Método de B")

class C(B):
    def metodo3(self):
        print("Método de C")
```

Salida:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    class A(C):
NameError: name 'C' is not defined
```

Intentar hacer herencia circular de otras formas sólo terminará sobrescribiendo clases, haciendo que siempre haya una superclase de las otras 2.

19.10. 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.

```
class A:
    def metodo(self):
        print("A")

class B(A):
    def metodo(self):
        print("B")

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

b = B()

b.metodo()
b.metodo_super()
```

Salida:

```
B
A
```

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

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

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

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

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

h = Humano("Javier", 20)
h.presentarse()

print()

t = Trabajador("John", 30, "profesor")
t.presentarse()
```

Salida:

```
Hola, soy Javier
Tengo 20 años

Hola, soy John
Tengo 30 años
Soy profesor
```

Otra forma de verlo es que cuando una subclase no tiene un método, lo buscará en su superclase antes de lanzar una excepción diciendo que no existe. Si no lo encuentra en su superclase, lo buscará en la superclase de su superclase, y así sucesivamente.

El método usado será el primero que encuentre al realizar esa búsqueda.

19.11. 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.

19.12. Sobrecarga de operadores aritméticos

Un uso común de métodos mágicos es la sobrecarga de operadores. Esto significa definir operadores para clases personalizadas que permiten que operadores tales como `+` y `*` sean utilizados en ellas.

El método mágico `__add__()` permite sobrecargar el operador `+`, lo cual permite darle un comportamiento personalizado.

```
class Vector2D:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, otro):
        return Vector2D(self.x + otro.x, self.y + otro.y)

v1 = Vector2D(5, 7)
v2 = Vector2D(3, 9)
resultado = v1 + v2

print(resultado.x)
print(resultado.y)
```

Salida:

```
8
16
```

El método `__add__()` suma los atributos correspondientes de los objetos y devuelve un nuevo objeto que contiene el resultado. Una vez definido, se pueden sumar dos objetos de una clase entre sí.

Otra forma de verlo es que el intérprete de Python siempre interpretará el operador `+` como el método `__add__()` de su clase.

Los métodos mágicos para operadores comunes son:

```

Operador +: __add__()
Operador -: __sub__()
Operador *: __mul__()
Operador /: __truediv__()
Operador //: __floordiv__()
Operador %: __mod__()
Operador **: __pow__()
Operador &: __and__()
Operador ^: __xor__()
Operador |: __or__()

```

Y después el intérprete de Python interpreta los operadores así:

```

# Si x e y son del mismo tipo y este tipo tiene implementado los métodos mágicos
x = 10
y = 5

resultado = [
    x + y == x.__add__(y),
    x - y == x.__sub__(y),
    x * y == x.__mul__(y),
    x / y == x.__truediv__(y),
    x // y == x.__floordiv__(y),
    x % y == x.__mod__(y),
    x ** y == x.__pow__(y),
    x & y == x.__and__(y),
    x ^ y == x.__xor__(y),
    x | y == x.__or__(y)
]

print(resultado)

```

Salida:

```
[True, True, True, True, True, True, True, True, True, True]
```

A continuación, se muestra un ejemplo de implementación del método `__truediv__`.

```

class CadenaEspecial:
    def __init__(self, contenido):
        self.contenido = contenido

    def __truediv__(self, otro):
        linea = "=" * len(otro.contenido)
        return "\n".join([self.contenido, linea, otro.contenido])

palabra1 = CadenaEspecial("hola")
palabra2 = CadenaEspecial("mundo")

print(palabra1 / palabra2)

```

Salida:

```

hola
=====
mundo

```

La sobrecarga de operadores no necesita que los métodos mágicos nuevos cumplan una función parecida a la original. Pueden ser cualquier cosa.

19.13. Métodos mágicos reversos

Un caso particular ocurre cuando “x” e “y” son de tipos distintos y “x” no tiene definido un método mágico, por ejemplo `__add__()`. En este caso, Python intentará calcular la operación al revés, la cual en este caso será `y + x`. Para realizar esta operación llamará al método reverso `__radd__()` de `y`.

Hay métodos equivalentes para todos los operadores mencionados anteriormente.

Si x e y son de tipo distinto y x no ha implementado los métodos mágicos:

```
x + y -> y + x -> y.__radd__(x)
x - y -> y - x -> y.__rsub__(x)
x * y -> y * x -> y.__rmul__(x)
x / y -> y / x -> y.__rtruediv__(x)
x // y -> y // x -> y.__rfloordiv__(x)
x % y -> y % x -> y.__rmod__(x)
x ** y -> y ** x -> y.__rpow__(x)
x & y -> y & x -> y.__rand__(x)
x ^ y -> y ^ x -> y.__rxor__(x)
x | y -> y | x -> y.__ror__(x)
```

Y un ejemplo de implementación de métodos reversos:

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

class Gato:
    def __init__(self, nombre):
        self.nombre = nombre

    def __radd__(self, otro):
        # Perro + Gato = string
        return "\t".join([self.nombre, otro.nombre])

perro = Perro("Joe")
gato = Gato("Ollie")

print(perro + gato)
```

Salida:

```
Ollie  Joe
```

19.14. Sobrecarga de operadores de comparación

Python también ofrece métodos mágicos para comparaciones.

```
Operador < (less than): __lt__()
Operador <= (less or equal): __le__()
Operador == (equal): __eq__()
Operador != (not equal): __ne__()
Operador > (greater than): __gt__()
Operador >= (greater or equal): __ge__()
```

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

```
class CadenaEspecial:
    def __init__(self, contenido):
        self.contenido = contenido

    def __gt__(self, otro):
        for i in range(len(otro.contenido) + 1):
            resultado = otro.contenido[:i] + ">" + self.contenido
            resultado += ">" + otro.contenido[i:]
            print(resultado)

palabra1 = CadenaEspecial("hola")
palabra2 = CadenaEspecial("mundo")

palabra1 > palabra2
```

Salida:

```
>hola>mundo
m>hola>undo
mu>hola>ndo
mun>hola>do
mund>hola>o
mundo>hola>
```

19.15. Métodos mágicos de contenedores

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

```
Método len(): __len__()
Indexación: __getitem__()
Asignar valores indexados: __setitem__()
Borrar valores indexados: __delitem__()
Iteración sobre objetos (por ejemplo en bucles for): __iter__()
Operador in: __contains__()
```

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

```

import random

class ListaImprecisa:
    def __init__(self, contenido):
        self.contenido = contenido

    def __getitem__(self, indice):
        return self.contenido[indice + random.randint(-1, 1)]

    def __len__(self):
        return random.randint(0, len(self.contenido) * 2)

lista_imprecisa = ListaImprecisa(["A", "B", "C", "D", "E"])

# Los resultados son aleatorios, haciendo que la lista sea imprecisa
print(len(lista_imprecisa))
print(len(lista_imprecisa))
print(len(lista_imprecisa))
print(lista_imprecisa[2])
print(lista_imprecisa[2])
print(lista_imprecisa[2])

```

19.16. Ciclo de vida de un objeto

El ciclo de vida de un objeto está conformado por su creación, manipulación y destrucción.

La primera etapa del ciclo de vida de un objeto es la definición de la clase a la cual pertenece.

La siguiente etapa es la instanciación de un objeto, cuando el método `__init__()` es llamado. La memoria es asignada para almacenar la instancia. Justo antes de que esto ocurra, el método `__new__()` de la clase es llamado, para asignar la memoria necesaria. Este es normalmente redefinido sólo en casos especiales.

Luego de que ocurra lo anterior el objeto estará listo para ser utilizado.

Otro código puede interactuar con el objeto, llamando sus métodos o accediendo a sus atributos. Eventualmente, terminará de ser utilizado y podrá ser destruido.

Cuando un objeto es destruido, la memoria asignada se libera y puede ser utilizada para otros propósitos.

La destrucción de un objeto ocurre cuando su contador de referencias llega a cero. La cuenta de referencias es el número de variables y otros elementos que se refieren al objeto.

Si nada se está refiriendo al objeto (tiene una cuenta de referencias de 0) nada puede interactuar con este, así que puede ser eliminado con seguridad.

En algunas situaciones, dos (o más) objetos pueden solo referirse entre ellos, y por lo tanto pueden ser eliminados también.

La sentencia `del` reduce la cuenta de referencias de un objeto por 1, y a menudo conlleva a su eliminación. El método mágico de la sentencia `del` es `__del__()`.

El proceso de eliminación de objetos cuando ya no son necesarios se denomina recolección de basura (garbage collection).

En resumen, el contador de referencias de un objeto se incrementa cuando se le es asignado un nuevo nombre o es colocado en un contenedor (una lista, tupla o diccionario). La cuenta de referencias de un objeto se disminuye cuando es eliminado con `del`, su referencia es reasignada, o su referencia sale fuera del alcance. Cuando la cuenta de referencias de un objeto llega a 0, Python lo elimina automáticamente.

```

a = 42 # Creación del objeto <42>
b = a # Aumenta el contador de referencias de <42>
c = [a] # Aumenta el contador de referencias de <42>

del a # Reduce el contador de referencias de <42>
b = 100 # Reduce el contador de referencias de <42>
c[0] = -1 # Reduce el contador de referencias de <42>

# <42> ya puede ser eliminado

```

Lenguajes de bajo nivel como C no tienen esta clase de manejo de memoria automático. Se debe realizar manualmente en ellos.

19.17. 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 responsables 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.

19.18. Métodos y atributos privados

Los métodos y atributos débilmente privados tienen un único guión bajo al principio. Esto señala que son privados, y no deberían ser utilizados por código externo.

```

class Cola:
    def __init__(self, contenido):
        self._lista_oculta = list(contenido)

    def agregar(self, valor):
        self._lista_oculta.insert(0, valor)

    def quitar(self):
        return self._lista_oculta.pop()

    def __repr__(self):
        return "Cola({})".format(self._lista_oculta)

cola = Cola([1, 2, 3])
print(cola)

cola.agregar(0)
print(cola)

cola.quitar()
print(cola)

# nada evita que se pueda acceder al atributo débilmente privado
print(cola._lista_oculta)

```


Salida:

```
Cola([1, 2, 3])
Cola([0, 1, 2, 3])
Cola([0, 1, 2])
[0, 1, 2]
```

Sin embargo, es en su mayor parte sólo una convención, y no impide que el código externo los acceda. Su único efecto verdadero es que `from nombre_de_modulo import *` no importará a las variables que empiecen por un único guión bajo.

Los métodos y atributos fuertemente privados tienen doble guión bajo al principio de sus nombres. Esto hace que sus nombres sean decorados, lo cual significa que no pueden ser accedidos desde afuera de la clase.

El propósito de esto no es asegurar que se mantengan privados, sino de evitar errores en caso de que existan subclases que tengan métodos o atributos con los mismos nombres.

Los métodos con nombres decorados pueden aún ser accedidos externamente pero por un nombre diferente. Por ejemplo, método `__privado()` de la clase `Clase` puede ser accedido externamente con `_Clase__metodoprivado()`.

```
class Spam:
    __egg = 7

    def mostrar_egg(self):
        print(self.__egg)

s = Spam()

s.mostrar_egg()
print(s.__Spam__egg)
print(s.__egg)
```

Salida:

```
7
7
Traceback (most recent call last):
  File "c:\Users\vnico\Desktop\Resumen Python
    ↳ 3\codigo\20-objetos\fuertemente_privado.py", line 11, in <module>
    print(s.__egg)
AttributeError: 'Spam' object has no attribute '__egg'
```

19.19. Métodos de clase

Hasta ahora, sólo se han visto métodos que son llamados por una instancia de clase, la cual es pasada al parámetro `self` del método.

Los métodos de clase son distintos: son llamados por una clase que es pasada al parámetro `cls` del método, el cual reemplaza a `self`.

Un uso común de estos son los métodos de fábrica, los cuales instancian un objeto de la clase utilizando diferentes parámetros que aquellos normalmente pasados al constructor de la clase.

Los métodos de clase son marcados con el decorador `classmethod`.

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

    def calcular_area(self):
        return self.ancho * self.altura

    @classmethod
    def crear_cuadrado(cls, lado):
        return cls(lado, lado)

# Crea un cuadrado de lado 5
cuadrado = Rectangulo.crear_cuadrado(5)

print(cuadrado.calcular_area())
```

Salida:

```
25
```

19.20. Métodos estáticos

Los métodos estáticos son parecidos a los métodos de clase, excepto que no reciben argumentos adicionales como `self` o `cls`. Son idénticos a funciones regulares que pertenecen a una clase.

Son marcados con el decorador `staticmethod`.

```
class Pizza:
    def __init__(self, toppings):
        self.toppings = toppings

    @staticmethod
    def validar_topping(topping):
        if topping == "piña":
            raise ValueError("¡No se aceptan piñas!")
        else:
            return True

ingredientes = ["queso", "cebollas", "spam"]

if all(Pizza.validar_topping(i) for i in ingredientes):
    pizza = Pizza(ingredientes)
```

Los métodos estáticos se comportan como funciones normales, excepto por el hecho de que se pueden llamar desde una instancia de una clase. Esto significa que se pueden sin necesidad de instanciar un objeto de la clase.

19.21. Propiedades

Las propiedades ofrecen una forma de personalizar el acceso a los atributos de instancia.

Son creadas al poner el decorador `property` encima de un método, lo cual significa que cuando el atributo de instancia con el mismo nombre que el método es accedido, el método será llamado en su lugar.

Un uso común de una propiedad es hacer que un atributo sea de solo lectura.

```

class Pizza:
    def __init__(self, toppings):
        self.toppings = toppings

    @property
    def pinia_permitida(self):
        return False

pizza = Pizza(["queso", "tomate"])

print(pizza.pinia_permitida)

# No puede cambiarse el atributo
pizza.pinia_permitida = True

```

Salida:

```

False
Traceback (most recent call last):
  File "c:\Users\vnico\Desktop\Resumen Python 3\codigo\20-objetos\propiedad_1.py",
    ↪ line 12, in <module>
        pizza.pinia_permitida = True
AttributeError: can't set attribute

```

```

class Persona:
    def __init__(self, edad):
        self.edad = int(edad)

    @property
    def es_adulto(self):
        return self.edad >= 18

```

Las propiedades también pueden ser fijadas definiendo funciones de tipo getter y setter.

Un getter toma el valor de una propiedad, mientras el setter fija el valor.

Para definir un getter, se debe usar un decorador con el mismo nombre de la propiedad, seguido por **.getter**. Lo mismo ocurre para un setter, pero va seguido por **.setter**.

```

class Pizza:
    def __init__(self, toppings):
        self.toppings = toppings
        self.pinia_permitida = False

    @property
    def pinia_permitida(self):
        return self.pinia_permitida

    @pinia_permitida.setter
    def pinia_permitida(self, valor):
        if valor:
            contrasena = input("Ingrese la contraseña: ")
            if contrasena == "1234":
                self.pinia_permitida = valor
            else:
                raise ValueError("Alerta, ¡un intruso!")

pizza = Pizza(["queso", "tomate"])
print(pizza.pinia_permitida)

pizza.pinia_permitida = True
print(pizza.pinia_permitida)

```

Capítulo 20

Expresiones regulares

20.1. Expresiones regulares

Las expresiones regulares son una potente herramienta para varias clases de manipulación de cadenas.

Son un lenguaje de dominio específico (DSL, por sus siglas en inglés) que está presente como una biblioteca en la mayoría de los lenguajes de programación, no solo Python.

Son útiles para 2 tareas principales:

- Verificar que las cadenas correspondan a un patrón (por ejemplo, que una cadena tenga el formato de un correo electrónico).
- Realizar substitutiones en una cadena (como por ejemplo, cambiar la ortografía del inglés estadounidense al inglés británico).

Los lenguajes de dominio específico son mini-lenguajes de programación altamente especializados.

Las expresiones regulares son un ejemplo popular y SQL (para la manipulación de datos) es otro.

Lenguajes privados específicos del dominio son a menudo usados para propósitos industriales particulares.

Para usar expresiones regulares en Python, se debe importar el módulo `re` que pertenece a la biblioteca estándar.

```
import re
```

20.2. Definición de expresiones regulares

Para definir una expresión regular, se puede almacenar en una variable y el patrón se escribe como un string. Se recomienda usar cadenas puras “raw string”, las cuales empiezan por la letra `r` fuera de las comillas.

```
import re

patron = r"hola"

print(patron)
print(type(patron))
```

Salida:

```
hola
<class 'str'>
```

Las cadenas puras ignoran las secuencias de escape propias de las cadenas de Python, pero no las de expresiones regulares. Es por este motivo que hace más fácil el uso de expresiones regulares.

20.3. Coincidencias parciales

Para revisar si una cadena encaja con un patrón, se puede usar el método `match()` para compararla con una expresión regular.

```
import re

patron = r"hola"

print(re.match(patron, "hola"))
print(re.match(patron, "hola mundo"))
print(re.match(patron, "ola"))
print(re.match(patron, "chao hola"))
```

Salida:

```
<re.Match object; span=(0, 4), match='hola'>
<re.Match object; span=(0, 4), match='hola'>
None
None
```

En caso de arriba, revisa todas las palabras que empiezan por `"hola "`. Si no se parecen al patrón, mostrará `None`.

Puede ser más conveniente convertir el resultado `match` a un booleano usando `bool()`.

```
import re

patron = r"hola"

print(bool(re.match(patron, "hola")))
print(bool(re.match(patron, "hola mundo")))
print(bool(re.match(patron, "ola")))
print(bool(re.match(patron, "chao hola")))
```

Salida:

```
True
True
False
False
```

Este resultado se parece al que se obtendría usando el método `startswith()`, pero la gracia de las expresiones regulares es que permiten patrones mucho más complejos, que se verán más tarde.

Otra opción es usar la función `search()`, que busca si un patrón aparece en cualquier parte de la cadena, no solo al principio.

```
import re

patron = r"hola"

print(re.search(patron, "hola"))
print(re.search(patron, "hola mundo"))
print(re.search(patron, "ola"))
print(re.search(patron, "chao hola"))
```

Salida:

```
<re.Match object; span=(0, 4), match='hola'>
<re.Match object; span=(0, 4), match='hola'>
None
<re.Match object; span=(5, 9), match='hola'>
```

20.4. Coincidencias múltiples

La función `findall()` devuelve una lista con todas las subcadenas que coinciden con un patrón.

```
import re

patron = r"hola"

print(re.findall(patron, "hola mundo, chao"))
print(re.findall(patron, "holachaoholachaohola"))
```

Salida:

```
['hola']
['hola', 'hola', 'hola']
```

Si se combina `findall()` con `len()`, se puede ver la cantidad de ocurrencias de un patrón.

```
import re

patron = r"10"

print(len(re.findall(patron, "1001010101010010101101")))
print(len(re.findall(patron, "01010100011100")))
```

Salida:

```
9
4
```

También existe la función `finditer()` que hace lo mismo que `findall()` pero devuelve un iterador en vez de una lista.

```
import re

patron = r"hola"

print(re.finditer(patron, "hola mundo, chao"))
print(re.finditer(patron, "holachaoholachaohola"))
```

Salida:

```
<callable_iterator object at 0x000002BBA61DDF10>
<callable_iterator object at 0x000002BBA61DDF10>
```

20.5. Detalles sobre los resultados

La búsqueda regex devuelve un objeto con varios métodos que dan detalles sobre ésta.

Estos métodos incluyen `group()` que devuelve la cadena que coincide, `start()` que devuelve la posición inicial de esta cadena, `end()` que devuelve la posición final y `span()` que devuelve la posición inicial y final como una tupla.

```
import re

patron = r"pam"
resultado = re.search(patron, "foobarspameggs")

if resultado:
    print(resultado.group())
    print(resultado.start())
    print(resultado.end())
    print(resultado.span())
```

Salida:

```
pam
7
10
(7, 10)
```

20.6. Buscar y reemplazar

Uno de los métodos de `re` más importantes que utiliza expresiones regulares es `sub()`.

Este método reemplaza todas las ocurrencias de un patrón con un string dado. Devuelve una cadena modificada.

```
import re

patron = r"10"
reemplazo = "*"

print(re.sub(patron, reemplazo, "10010101010010101101"))
print(re.sub(patron, reemplazo, "01010100011100"))
```

Salida:

```
*0*****0*1*1
0***0011*0
```

Se le puede entregar el argumento `count`, que especifica la cantidad de reemplazos que se quieren.

```
import re

for i in range(1, 8):
    print(re.sub(r"10", "*", "101010101010", count=i))
```

Salida:


```
*101010101010
**1010101010
***10101010
****101010
*****1010
*****10
*****
*****
```

El valor predeterminado de **count** es 0, lo que significa que se reemplazarán todas las coincidencias.

```
import re

palabra = "3733234233432348393333133"

print(re.sub(r"3", "", palabra))
print(re.sub(r"3", "", palabra, count=0))
```

Salida:

```
7242424891
7242424891
```

20.7. Coincidencias exactas

La función **fullmatch()** compara de forma más estricta que **match()**, debido a que toma en cuenta toda la cadena con la que se compara el patrón.

```
import re

patron = r"hola"

# Coincidencias parciales
print(re.match(patron, "hola"))
print(re.match(patron, "hola mundo"))

# Coincidencias exactas
print(re.fullmatch(patron, "hola"))
print(re.fullmatch(patron, "hola mundo"))
```

Salida:

```
<re.Match object; span=(0, 4), match='hola'>
<re.Match object; span=(0, 4), match='hola'>
<re.Match object; span=(0, 4), match='hola'>
None
```

20.8. Metacaracteres

Los metacaracteres son los que hacen a las expresiones regulares más potentes que los métodos de cadena normales. Permiten crear patrones que representan conceptos como “una o más repeticiones de una vocal”.

Todos los metacaracteres son los siguientes:

. ^ \$ * + ? { } [] () \ |

La existencia de los metacaracteres presenta un problema si se desea crear una expresión regular que corresponda a un metacarácter literal, como por ejemplo "\$". La solución a estos casos es colocar una barra diagonal invertida \ antes de dicho carácter.

Sin embargo, esto también tendría problemas porque en Python las barras diagonales \ tienen la función de iniciar secuencias de escape dentro de strings. Por esta razón es que se recomienda usar cadenas puras en las expresiones regulares.

```
import re

# Para evitar tener que escribir esto
patron1 = "\\$"

# Se usan cadenas puras
patron2 = r"\"$\"

print(re.match(patron1, "$"))
print(re.match(patron2, "$"))

# Para expresiones muy largas resulta especialmente útil
patron3 = r"\"$\"\\.\"?\"*\"+\"

# En vez de
patron4 = "\\$\"\\.\"?\"*\"+\"

print(re.match(patron3, "$.*+"))
print(re.match(patron4, "$.*+"))
```

Salida:

```
<re.Match object; span=(0, 1), match='$'>
<re.Match object; span=(0, 1), match='$'>
<re.Match object; span=(0, 5), match='$.*+'>
<re.Match object; span=(0, 5), match='$.*+'>
```

20.9. Punto

El metacarácter más sencillo es el punto ., que coincide con cualquier carácter excepto una nueva línea.

```
import re

patron = r"color"

print(re.fullmatch(patron, "color"))
print(re.fullmatch(patron, "calor"))
print(re.fullmatch(patron, "collar"))
```

Salida:

```
<re.Match object; span=(0, 5), match='color'>
<re.Match object; span=(0, 5), match='calor'>
None
```

```
import re

patron = r"hol.a.mundo"

print(re.fullmatch(patron, "hola\nmundo"))
```

Salida:

```
None
```

Básicamente, el punto `.` actúa como un “comodín”.

Otro uso puede ser para pedir palabras de un largo igual a la cantidad de puntos.

```
import re

patron = r"....." # cadenas de 5 caracteres

print(re.fullmatch(patron, "verde"))
print(re.fullmatch(patron, "12345"))
print(re.fullmatch(patron, "@!/&?"))

print(re.fullmatch(patron, ""))

# si se usa match(), serían de 5 o más caracteres
print(re.fullmatch(patron, "qwertyuiopasdfghjklzxcvbnm"))
print(re.match(patron, "qwertyuiopasdfghjklzxcvbnm"))
```

Salida:

```
<re.Match object; span=(0, 5), match='verde'>
<re.Match object; span=(0, 5), match='12345'>
<re.Match object; span=(0, 5), match='@!/&?'>
None
None
<re.Match object; span=(0, 5), match='qwert'>
```

20.10. Inicio y final

Los metacaracteres `^` y `$` coinciden con el inicio y el final de una cadena, respectivamente. `^` busca palabras que comiencen con el carácter que le sigue y `$` busca palabras que terminen con el carácter que le precede.

En los ejemplos se usa `search()` porque `match()` busca desde el principio del string, lo que sería redundante con `^`.

```
import re

# palabras que empiezan con s
patron = r"^s"

print(re.match(patron, "sandía"))
print(re.match(patron, "saludar"))
print(re.match(patron, "final"))
print(re.match(patron, "campana"))

# a fullmatch() le falta información para poder aceptar todos los casos
print(re.fullmatch(patron, "sandía"))
print(re.fullmatch(patron, "saludar"))
print(re.fullmatch(patron, "final"))
print(re.fullmatch(patron, "campana"))
```

Salida:

```
<re.Match object; span=(0, 1), match='s'>
<re.Match object; span=(0, 1), match='s'>
None
None
None
None
None
None
```

```
import re

# palabras que terminan con a
patron = r"a$"

print(re.search(patron, "sandía"))
print(re.search(patron, "saludar"))
print(re.search(patron, "final"))
print(re.search(patron, "campana"))
```

Salida:

```
<re.Match object; span=(5, 6), match='a'>
None
None
<re.Match object; span=(6, 7), match='a'>
```

[^] y ^{\$} por si solos no son suficiente para buscar palabras que comiencen con un carácter y terminen con otro (al mismo tiempo).

20.11. Clases de caracteres

Los corchetes `[]` son metacaracteres usados para definir clases de caracteres, que son conjuntos de caracteres aceptados. El patrón solo necesita que aparezca uno de los caracteres dentro del grupo.

```
import re

patron = r"[aeiou]" # cualquier cadena con vocales minúsculas

print(re.search(patron, "Python"))
print(re.search(patron, "C"))
print(re.search(patron, "Java"))
print(re.search(patron, "C++"))
```

Salida:

```
<re.Match object; span=(4, 5), match='o'>
None
<re.Match object; span=(1, 2), match='a'>
None
```

```
import re

patron = r"[aeiou]" # cualquier cadena con vocales minúsculas

print(re.findall(patron, "Python es un lenguaje de programación"))
```

Salida:

```
['o', 'e', 'u', 'e', 'u', 'a', 'e', 'e', 'o', 'a', 'a', 'i']
```

Si se ingresan en sucesión, el patrón acepta un carácter del primer conjunto en orden seguido de uno del segundo.

```
import re

patron = r"[abc][def]" # a, b o c seguida de d, e o f

print(re.findall(patron, "abcdef ac ab de cd ce be fb bf be cf fa"))
```

Salida:

```
['cd', 'cd', 'ce', 'be', 'bf', 'be', 'cf']
```

20.12. Rangos de caracteres

Dentro de una clase de caracteres se pueden especificar rangos para evitar tener que escribir muchas letras o números.

Las más usadas son las siguientes:

- **[a-z]**: Para cualquier letra del alfabeto minúscula.
- **[A-Z]**: Para cualquier letra del alfabeto mayúscula.
- **[0-9]**: Para cualquier dígito.

```
import re

print(re.findall(r"[a-z]", "Palabra"))
print(re.findall(r"[A-Z]", "PascalCase"))
print(re.findall(r"[0-9]", "abc123"))
```

Salida:

```
['a', 'l', 'a', 'b', 'r', 'a']
['P', 'C']
['1', '2', '3']
```

Los rangos se pueden combinar dentro de una misma clase.

```
import re

print(re.findall(r"[A-Za-z]", "PascalCase"))
print(re.findall(r"[0-9A-Z]", "HOLA123"))
```

Salida:

```
['P', 'a', 's', 'c', 'a', 'l', 'C', 'a', 's', 'e']
['H', 'O', 'L', 'A', '1', '2', '3']
```

Para representar dígitos, también se puede usar `\d`.

```
import re

print(re.findall(r"\d", "123"))
```

Salida:

```
['1', '2', '3']
```

Los rangos se pueden limitar dependiendo de lo que se necesite.

```
import re

print(re.findall(r"[j-s]", "abcdefghijklmnopqrstuvwxy"))

# Lo contrario al anterior
print(re.findall(r"[a-it-z]", "abcdefghijklmnopqrstuvwxy"))

print(re.findall(r"[1-5]", "1234567890"))

print(re.findall(r"[1-5][0-9]", "4, 5, 10, 19, 25, 44, 49, 59, 60, 75, 92"))
```

Salida:

```
['j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's']
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 't', 'u', 'v', 'w', 'x', 'y', 'z']
['1', '2', '3', '4', '5']
['10', '19', '25', '44', '49', '59']
```

20.13. Clases invertidas

Si se ingresa el metacarácter `^` al principio de una clase de caracteres, invierte todos los contenidos de la clase.

```
import re

patron = r"^[A-Za-z0-9]"

# Sólo símbolos o letras con acentos extraños
print(re.findall(patron, "abcDEF789"))
print(re.findall(patron, "#.!°?[^, _%&$|*+]"))
print(re.findall(patron, "áéíóú"))
```

Salida:

```
[ ]
['#', '.', '!', '°', '?', ')', '^', '[', ',', '-', '_', '%', '&', '$', '|', '*', '+']
['á', 'é', 'í', 'ó', 'ú']
```

Esto solo funciona si se coloca al principio, en cualquier otra posición no tiene función.

20.14. Repeticiones

Los metacaracteres `*`, `+`, `?`, `{` y `}` se usan para especificar repeticiones de caracteres.

El metacarácter `*` significa "0 o más repeticiones de lo anterior". También se conoce como el identificador del conjunto cierre en ciencias de computación.

```
import re

patron = r"V*" # 0 más letras V

print(re.fullmatch(patron, "V"))
print(re.fullmatch(patron, "VV"))
print(re.fullmatch(patron, "VVVVVV"))
print(re.fullmatch(patron, ""))
print(re.fullmatch(patron, "B"))

print(re.fullmatch(patron, "VVVVBBBB"))
print(re.fullmatch(r"V*B*", "VVVVBBBB"))
```

Salida:

```
<re.Match object; span=(0, 1), match='V'>
<re.Match object; span=(0, 2), match='VV'>
<re.Match object; span=(0, 6), match='VVVVVV'>
<re.Match object; span=(0, 0), match=''>
None
None
<re.Match object; span=(0, 8), match='VVVVBBBB'>
```

Se puede usar `*` (y cualquier otro metacarácter de repetición) después de clases de caracteres.

```
import re

patron = r"[A-Z0-9]*" # cualquier cantidad de números y letras mayúsculas

print(re.fullmatch(patron, "ABC123"))
print(re.fullmatch(patron, "abc123"))
print(re.fullmatch(patron, ""))

print(re.fullmatch(patron, "A"))
print(re.fullmatch(patron, "abcd"))
print(re.fullmatch(patron, "123456"))
```

Salida:

```
<re.Match object; span=(0, 6), match='ABC123'>
None
<re.Match object; span=(0, 0), match=''>
<re.Match object; span=(0, 1), match='A'>
None
<re.Match object; span=(0, 6), match='123456'>
```

El metacarácter **+** es muy similar a ***** excepto que significa “1 o más repeticiones de lo anterior”, es decir, no permite que no aparezca el carácter anterior. También se conoce como cierre positivo.

```
import re

patron = r"g+" # 1 o más letras g

print(re.fullmatch(patron, ""))
print(re.fullmatch(patron, "g"))
print(re.fullmatch(patron, "gg"))
print(re.fullmatch(patron, "gggggggggggggggggg"))
```

Salida:

```
None
<re.Match object; span=(0, 1), match='g'>
<re.Match object; span=(0, 2), match='gg'>
<re.Match object; span=(0, 18), match='gggggggggggggggggg'>
```

El metacarácter **?** indica que el carácter anterior es opcional, lo que significa que puede aparecer 0 o 1 vez.


```
import re

patron = r"a?b?c?" # a opcional, b opcional, c opcional

print(re.fullmatch(patron, ""))

print(re.fullmatch(patron, "a"))
print(re.fullmatch(patron, "b"))
print(re.fullmatch(patron, "c"))

print(re.fullmatch(patron, "ab"))
print(re.fullmatch(patron, "ac"))
print(re.fullmatch(patron, "bc"))

print(re.fullmatch(patron, "ba"))
print(re.fullmatch(patron, "ca"))
print(re.fullmatch(patron, "cb"))

print(re.fullmatch(patron, "abc"))

print(re.fullmatch(patron, "cba"))
```

Salida:

```
<re.Match object; span=(0, 0), match=''>
<re.Match object; span=(0, 1), match='a'>
<re.Match object; span=(0, 1), match='b'>
<re.Match object; span=(0, 1), match='c'>
<re.Match object; span=(0, 2), match='ab'>
<re.Match object; span=(0, 2), match='ac'>
<re.Match object; span=(0, 2), match='bc'>
None
None
None
<re.Match object; span=(0, 3), match='abc'>
None
```

Las llaves { y } son metacaracteres que permiten indicar la cantidad de repeticiones de algo.

```
import re

patron = r"9{1,3}" # entre 1 y 3 números 9

print(re.fullmatch(patron, ""))
print(re.fullmatch(patron, "9"))
print(re.fullmatch(patron, "99"))
print(re.fullmatch(patron, "999"))
print(re.fullmatch(patron, "9999"))
```

Salida:

```
None
<re.Match object; span=(0, 1), match='9'>
<re.Match object; span=(0, 2), match='99'>
<re.Match object; span=(0, 3), match='999'>
None
```

Se debe recordar no insertar ningún espacio después de la coma.

Decir `{0,1}` quiere decir entre 0 y 1 repeticiones del caracter anterior, lo que es equivalente a `?`.

```
import re

print(re.fullmatch(r"9{0,1}" , ""))
print(re.fullmatch(r"9{0,1}" , "9"))
print(re.fullmatch(r"9{0,1}" , "99"))

print(re.fullmatch(r"9?" , ""))
print(re.fullmatch(r"9?" , "9"))
print(re.fullmatch(r"9?" , "99"))
```

Salida:

```
<re.Match object; span=(0, 0), match=''>
<re.Match object; span=(0, 1), match='9'>
None
<re.Match object; span=(0, 0), match=''>
<re.Match object; span=(0, 1), match='9'>
None
```

Si se omite el segundo caracter, se considera como “hasta infinito”, lo que significa que `{0,}` es equivalente a `*` y `{1,}` es equivalente a `+`.

```
import re

print(re.fullmatch(r"9{0,}" , ""))
print(re.fullmatch(r"9{0,}" , "9"))
print(re.fullmatch(r"9{0,}" , "9999999999"))

print(re.fullmatch(r"9*" , ""))
print(re.fullmatch(r"9*" , "9"))
print(re.fullmatch(r"9*" , "9999999999"))
```

Salida:

```
<re.Match object; span=(0, 0), match=''>
<re.Match object; span=(0, 1), match='9'>
<re.Match object; span=(0, 10), match='9999999999'>
<re.Match object; span=(0, 0), match=''>
<re.Match object; span=(0, 1), match='9'>
<re.Match object; span=(0, 10), match='9999999999'>
```

```
import re

print(re.fullmatch(r"9{1,}" , ""))
print(re.fullmatch(r"9{1,}" , "9"))
print(re.fullmatch(r"9{1,}" , "9999999999"))

print(re.fullmatch(r"9+" , ""))
print(re.fullmatch(r"9+" , "9"))
print(re.fullmatch(r"9+" , "9999999999"))
```

Salida:

```

None
<re.Match object; span=(0, 1), match='9'>
<re.Match object; span=(0, 10), match='9999999999'>
None
<re.Match object; span=(0, 1), match='9'>
<re.Match object; span=(0, 10), match='9999999999'>

```

Por último, usando el metacarácter `*` también se pueden verificar palabras que empiezan con un carácter y terminan con otro, al mismo tiempo.

```

import re

empieza_s = r"^s.*" # palabras que empiezan con s
termina_a = r"*.a$" # palabras que terminan con a
s_a = r"^s.*a$" # palabras que empiezan con s y terminan con a

# ahora ya funciona fullmatch()
print(re.fullmatch(empieza_s, "sandía"))
print(re.fullmatch(empieza_s, "saludar"))
print(re.fullmatch(empieza_s, "final"))
print(re.fullmatch(empieza_s, "campana"))
print()

print(re.fullmatch(termina_a, "sandía"))
print(re.fullmatch(termina_a, "saludar"))
print(re.fullmatch(termina_a, "final"))
print(re.fullmatch(termina_a, "campana"))
print()

print(re.fullmatch(s_a, "sandía"))
print(re.fullmatch(s_a, "saludar"))
print(re.fullmatch(s_a, "final"))
print(re.fullmatch(s_a, "campana"))

```

Salida:

```

<re.Match object; span=(0, 6), match='sandía'>
<re.Match object; span=(0, 7), match='saludar'>
None
None

<re.Match object; span=(0, 6), match='sandía'>
None
None
<re.Match object; span=(0, 7), match='campana'>

<re.Match object; span=(0, 6), match='sandía'>
None
None
None

```

20.15. Grupos de caracteres

Un grupo puede ser creado al colocar los caracteres pertenecientes a él entre paréntesis `()`.

Esto permite a una secuencia de caracteres ser pasada como un único argumento a metacaracteres como `*` y `?`.

```

import re

# cualquier cantidad de "spam" seguida de cualquier cantidad de "eggs"
patron = r"(spam)*(eggs)*"

print(re.fullmatch(patron, ""))
print(re.fullmatch(patron, "spam"))
print(re.fullmatch(patron, "eggs"))
print()

print(re.fullmatch(patron, "spameggs"))
print(re.fullmatch(patron, "spamspameggseggs"))
print(re.fullmatch(patron, "spamspamspamspameggs"))
print(re.fullmatch(patron, "spameggseggseggseggs"))
print()

print(re.fullmatch(patron, "spamegg"))
print(re.fullmatch(patron, "spameggssпам"))

```

Salida:

```

<re.Match object; span=(0, 0), match=''>
<re.Match object; span=(0, 4), match='spam'>
<re.Match object; span=(0, 4), match='eggs'>

<re.Match object; span=(0, 8), match='spameggs'>
<re.Match object; span=(0, 16), match='spamspameggseggs'>
<re.Match object; span=(0, 20), match='spamspamspamspameggs'>
<re.Match object; span=(0, 20), match='spameggseggseggseggs'>

None
None

```

Capítulo 21

PEP

21.1. PEP

PEP proviene del inglés Python Enhancement Proposals, que significa Propuestas de Mejoras de Python. Son sugerencias para mejoras al lenguaje, hechas por desarrolladores experimentados de Python. El Zen de Python visto en el primer capítulo es el [PEP 20](#).

En este capítulo se verán algunos de los PEP más importantes, y un resumen sobre sus contenidos. La lista completa de PEPs puede encontrarse su [índice](#), donde se puede leer el contenido completo de cada uno.

21.2. PEP 8

El [PEP 8](#) es una guía de estilo sobre el tema de escritura de código legible. Contiene una serie de directrices en referencia a los nombres de variables, que pueden resumirse de esta forma.

- Los módulos deben tener nombres cortos y en minúsculas.

```
# Los módulos de la biblioteca estándar cumplen con esta regla
import math
import random
import time
import datetime
import itertools
import re
```

- Los nombres de clases deben estar en estilo **CamelCase**.

```
class TelefonoCelular:
    def __init__(self, numero):
        self.numero = numero
```

- La mayoría de nombres de variables y funciones deben estar en **snake_case**.

```
hola_mundo = "hola mundo"
pais_de_origen = "Chile"
```

```
def sumar_5(x):
    return x + 5
```

- Las constantes (variables que nunca cambian de valor) deben estar en **SCREAMING_SNAKE_CASE** también conocida como **CONSTANT_CASE**.

```
DIAS_DE_LA_SEMANA = 7
```

- Los nombres que causarían conflicto con las palabras claves de Python (como **class** o **if**) deben tener guiones bajos al principio.
- Se recomienda usar espacios entre operadores y después de escribir comas **,**.

```
print(20 + 30 - 50)
x = 44 * 22
y = 20 / 7
```

```
numeros = [1, 2, 3, 4, 5, 6]
idiomas = {"es": "español", "fr": "francés", "en": "inglés"}
```

- El espacio en blanco no debe ser abusado. Por ejemplo, se debe evitar poner espacios en blanco después de llaves o corchetes.

```
# Evitar hacer esto
colores = ( "rojo", "verde", "azul" )
aves = [     "paloma",     "tucán",     "gorrión"     , "águila"     ]
```

21.3. PEP 257

El [PEP 257](#) contiene convenciones sobre la redacción de docstrings para documentar código.

Capítulo 22

Empaquetamiento

Capítulo 23

Pruebas unitarias

23.1. Pruebas unitarias

23.2. Precedencia de operadores completa

En los capítulos anteriores, se mostró parte de la precedencia de operadores (por ejemplo, PEMDAS). En Python, todos los operadores tienen un orden específico en el que funcionan, incluyendo a muchos que no se han visto todavía.

El orden definitivo es el siguiente:

- 1) `()`: paréntesis para agrupar operaciones
- 2) `f(argumentos)`: llamadas a funciones
- 3) `x[indice:indice]`: cortes (slicing)
- 4) `x[indice]`: subscripción
- 5) `x.atributo`: referencias a atributos
- 6) `**`: exponenciación
- 7) `x`: NOT en bitwise
- 8) `+x`, `-x`: signo positivo y negativo
- 9) `*`, `/`, `//`, `%`: multiplicación, división, cociente y resto
- 10) `+`, `-`: adición y substracción (diferencia)
- 11) `<<`, `>>`: cambios en bitwise
- 12) `&`: AND en bitwise (unión)
- 13) `^`: XOR en bitwise (diferencia simétrica)
- 14) `|`: OR en bitwise (intersección)
- 15) `in`, `not in`, `is`, `is not`, `<`, `<=`, `>`, `>=`, `<>`, `!=`, `==`: comparaciones, membresía e identidad
- 16) `not x`: NOT booleano
- 17) `and`: AND booleano
- 18) `or`: OR booleano
- 19) `lambda`: expresiones lambda

Todas las operaciones que estén en un mismo nivel se realizarán de izquierda a derecha.

Lo ideal sería utilizar paréntesis para evitar cualquier confusión que pueda causar el uso de muchos operadores, pero también es importante conocer este orden para los casos en los que no se utilizaron.

Capítulo 24

Interfaz gráfica

Capítulo 25

Algoritmos de ordenamiento

25.1. Bubble sort

Pasos a seguir:

- 1) Se recorre una lista de izquierda a derecha.
- 2) Se compara cada par de valores adyacentes.
- 3) Si dicho par está desordenado (el primero es mayor que el segundo), los cambia de lugar.
- 4) Esto garantiza que en n iteraciones habrán n elementos ordenados al final de la lista.
- 5) Se repite hasta obtener la lista ordenada.

La forma más simple de programar este algoritmo nos deja con un algoritmo de $O(n^2)$.

```
def burbuja(lista):
    N = len(lista)

    for i in range(N - 1):
        for j in range(N - 1):
            if lista[j] > lista[j+1]:
                lista[j], lista[j+1] = lista[j+1], lista[j]

    return lista

numeros = [10, -1, -5, 100, 5, -0.5, -22, 44, -100, -10, 0.5, 0, -44, 1, 22]
print(burbuja(numeros))
```

Salida:

```
[-100, -44, -22, -10, -5, -1, -0.5, 0, 0.5, 1, 5, 10, 22, 44, 100]
```

Se puede ver el procedimiento de forma más detallada añadiendo un `print()` dentro del bucle `for`.

```
def burbuja(lista):
    N = len(lista)

    for i in range(N - 1):
        for j in range(N - 1):
            if lista[j] > lista[j+1]:
                lista[j], lista[j+1] = lista[j+1], lista[j]
                print(lista)

    return lista

numeros = [2, 8, 5, 3, 9, 4, 1]
burbuja(numeros)
```

Salida:

```
[2, 5, 8, 3, 9, 4, 1]
[2, 5, 3, 8, 9, 4, 1]
[2, 5, 3, 8, 4, 9, 1]
[2, 5, 3, 8, 4, 1, 9]
[2, 3, 5, 8, 4, 1, 9]
[2, 3, 5, 4, 8, 1, 9]
[2, 3, 5, 4, 1, 8, 9]
[2, 3, 4, 5, 1, 8, 9]
[2, 3, 4, 1, 5, 8, 9]
[2, 3, 1, 4, 5, 8, 9]
[2, 1, 3, 4, 5, 8, 9]
[1, 2, 3, 4, 5, 8, 9]
```

Para ordenar al revés sólo basta con invertir la condición dentro de la declaración **if**.

```
def burbuja(lista):
    N = len(lista)

    for i in range(N - 1):
        for j in range(N - 1):
            if lista[j] < lista[j+1]: # condición invertida
                lista[j], lista[j+1] = lista[j+1], lista[j]

    return lista

numeros = [10, -1, -5, 100, 5, -0.5, -22, 44, -100, -10, 0.5, 0, -44, 1, 22]
print(burbuja(numeros))
```

Salida:

```
[100, 44, 22, 10, 5, 1, 0.5, 0, -0.5, -1, -5, -10, -22, -44, -100]
```

25.2. Optimizaciones de bubble sort

Una pequeña optimización puede ser cambiar el **range** del segundo **for** por **range(N - 1 - i)**. Esto ocurre porque en cualquier iteración siempre habrán **i** elementos ordenados al final, los cuales se pueden omitir.

```
def burbuja(lista):
    N = len(lista)

    for i in range(N - 1):
        for j in range(N - 1 - i):
            if lista[j] > lista[j+1]:
                lista[j], lista[j+1] = lista[j+1], lista[j]

    return lista

numeros = [10, -1, -5, 100, 5, -0.5, -22, 44, -100, -10, 0.5, 0, -44, 1, 22]
print(burbuja(numeros))
```

Hacer eso sigue manteniendo el algoritmo en $O(n^2)$.

Otra optimización es añadir un booleano que revise si han habido cambios en cada iteración. Si en una iteración no hay cambios, significa que ya está ordenado y no es necesario seguir.

```
def burbuja(lista):
    N = len(lista)

    for i in range(N - 1):
        cambios = False

        for j in range(N - 1 - i):
            if lista[j] > lista[j+1]:
                lista[j], lista[j+1] = lista[j+1], lista[j]
                cambios = True

        if not cambios:
            break

    return lista

numeros = [10, -1, -5, 100, 5, -0.5, -22, 44, -100, -10, 0.5, 0, -44, 1, 22]
print(burbuja(numeros))
```

Esto hará que el algoritmo sea $O(n^2)$ y $\omega(n)$

25.3. Insertion sort

Pasos a seguir:

- 1) Se recorre una lista de izquierda a derecha.
- 2) Se compara cada elemento con los elementos a su izquierda, y se mueve a una posición adecuada si no está ordenado.
- 3) El primer ordenado siempre estará “ordenado”, ya que no tiene elementos a su izquierda.
- 4) Se repite hasta tener el último elemento ordenado.

La implementación más sencilla de este algoritmo es la siguiente. Tiene una complejidad de $O(n^2)$.

```
def insercion(lista):
    N = len(lista)

    for i in range(1, N):
        j = i

        while j > 0 and lista[j-1] > lista[j]:
            lista[j-1], lista[j] = lista[j], lista[j-1]

            j -= 1

    return lista

numeros = [10, -1, -5, 100, 5, -0.5, -22, 44, -100, -10, 0.5, 0, -44, 1, 22]
print(insercion(numeros))
```

Salida:

```
[-100, -44, -22, -10, -5, -1, -0.5, 0, 0.5, 1, 5, 10, 22, 44, 100]
```

Se puede añadir un método `print()` para ver en más detalle el procedimiento.

```
def insercion(lista):
    N = len(lista)

    for i in range(1, N):
        j = i

        while j > 0 and lista[j-1] > lista[j]:
            lista[j-1], lista[j] = lista[j], lista[j-1]
            print(lista)

            j -= 1

    return lista

numeros = [2, 8, 5, 3, 9, 4, 1]
insercion(numeros)
```

Salida:

```
[2, 5, 8, 3, 9, 4, 1]
[2, 5, 3, 8, 9, 4, 1]
[2, 3, 5, 8, 9, 4, 1]
[2, 3, 5, 8, 4, 9, 1]
[2, 3, 5, 4, 8, 9, 1]
[2, 3, 4, 5, 8, 9, 1]
[2, 3, 4, 5, 8, 1, 9]
[2, 3, 4, 5, 1, 8, 9]
[2, 3, 4, 1, 5, 8, 9]
[2, 3, 1, 4, 5, 8, 9]
[2, 1, 3, 4, 5, 8, 9]
[1, 2, 3, 4, 5, 8, 9]
```

Para ordenar al revés, sólo se debe invertir la condición dentro de la declaración `if`.

```
def insercion(lista):
    N = len(lista)

    for i in range(1, N):
        j = i

        while j > 0 and lista[j-1] < lista[j]: # condición invertida
            lista[j-1], lista[j] = lista[j], lista[j-1]

            j -= 1

    return lista

numeros = [10, -1, -5, 100, 5, -0.5, -22, 44, -100, -10, 0.5, 0, -44, 1, 22]
print(insercion(numeros))
```

Salida:

```
[100, 44, 22, 10, 5, 1, 0.5, 0, -0.5, -1, -5, -10, -22, -44, -100]
```

25.4. Selection sort

Pasos a seguir:

- 1) Se recorre una lista de izquierda a derecha.
- 2) La lista se “particiona” en 2 listas: uno ordenada (al principio) y otra desordenada (al final).
- 3) Se obtiene el elemento más pequeño de la partición desordenada y se mueve a la partición ordenada.
- 4) Se repite hasta que la partición desordenada esté vacía.

A continuación se muestra una implementación de este algoritmo. Tiene complejidad de $O(n^2)$.

```
def seleccion(lista):
    N = len(lista)

    for i in range(N - 1):
        i_minimo = i

        for j in range(i + 1, N):
            if lista[j] < lista[i_minimo]:
                i_minimo = j

        if i_minimo != i:
            lista[i], lista[i_minimo] = lista[i_minimo], lista[i]

    return lista

numeros = [10, -1, -5, 100, 5, -0.5, -22, 44, -100, -10, 0.5, 0, -44, 1, 22]
print(seleccion(numeros))
```

Salida:

```
[-100, -44, -22, -10, -5, -1, -0.5, 0, 0.5, 1, 5, 10, 22, 44, 100]
```

Para ver en detalle como funciona, se puede añadir el método `print()`.

```
def seleccion(lista):
    N = len(lista)

    for i in range(N - 1):
        i_minimo = i

        for j in range(i + 1, N):
            if lista[j] < lista[i_minimo]:
                i_minimo = j

        if i_minimo != i:
            lista[i], lista[i_minimo] = lista[i_minimo], lista[i]
            print(lista)

    return lista

numeros = [2, 8, 5, 3, 9, 4, 1]
seleccion(numeros)
```

Salida:

```
[1, 8, 5, 3, 9, 4, 2]
[1, 2, 5, 3, 9, 4, 8]
[1, 2, 3, 5, 9, 4, 8]
[1, 2, 3, 4, 9, 5, 8]
[1, 2, 3, 4, 5, 9, 8]
[1, 2, 3, 4, 5, 8, 9]
```

Al invertir la condición también se invertirá el orden.

```
def seleccion(lista):
    N = len(lista)

    for i in range(N - 1):
        i_minimo = i

        for j in range(i + 1, N):
            if lista[j] > lista[i_minimo]: # condición invertida
                i_minimo = j

        if i_minimo != i:
            lista[i], lista[i_minimo] = lista[i_minimo], lista[i]

    return lista

numeros = [10, -1, -5, 100, 5, -0.5, -22, 44, -100, -10, 0.5, 0, -44, 1, 22]
print(seleccion(numeros))
```

Salida:

```
[100, 44, 22, 10, 5, 1, 0.5, 0, -0.5, -1, -5, -10, -22, -44, -100]
```

25.5. Merge sort

Este algoritmo generalmente se implementa de forma recursiva. Se basa en “dividir para vencer”, dividiendo un problema complejo en varios más sencillos.

Pasos a seguir:

- 1) La lista se divide en mitades, hasta quedar con elementos individuales.
- 2) Las listas se agrupan por pares, y se empiezan a unir de vuelta.
- 3) Al unir las listas, se comparan los primeros elementos de cada una y se añaden a la lista unida. Después de añadirlos, se eliminan de su lista original.
- 4) Cuando una lista queda vacía, se añaden todos los elementos que queden en su lista pareja a la lista unida.
- 5) Se repite hasta llegar a la lista original, la cual quedará ordenada.

La siguiente implementación es de $O(n \log n)$.

```
def ordenar(lista):
    N = len(lista)

    if N == 1:
        return lista

    lista1 = lista[0:N // 2]
    lista2 = lista[N // 2:N]

    lista1 = ordenar(lista1)
    lista2 = ordenar(lista2)

    return unir(lista1, lista2)

def unir(a, b):
    c = []

    while len(a) != 0 and len(b) != 0:
        if a[0] > b[0]:
            c.append(b[0])
            b.pop(0)
        else:
            c.append(a[0])
            a.pop(0)

    # En este punto, alguna de las 2 listas está vacía
    # Se evita hacer operaciones con listas para mostrar mejor el algoritmo

    while len(a) != 0:
        c.append(a[0])
        a.pop(0)

    while len(b) != 0:
        c.append(b[0])
        b.pop(0)

    return c

numeros = [10, -1, -5, 100, 5, -0.5, -22, 44, -100, -10, 0.5, 0, -44, 1, 22]
print(ordenar(numeros))
```

Salida:

```
[-100, -44, -22, -10, -5, -1, -0.5, 0, 0.5, 1, 5, 10, 22, 44, 100]
```


Este algoritmo es complicado de entender, ya que usa mucho la recursión. Sin embargo, no es tan complejo como aparenta ser a primera vista.

Al igual que los algoritmos anteriores, sólo se necesita invertir una condición para invertir el resultado.

```
def ordenar(lista):
    N = len(lista)

    if N == 1:
        return lista

    lista1 = lista[0:N // 2]
    lista2 = lista[N // 2:N]

    lista1 = ordenar(lista1)
    lista2 = ordenar(lista2)

    return unir(lista1, lista2)

def unir(a, b):
    c = []

    while len(a) != 0 and len(b) != 0:
        if a[0] < b[0]: # condición invertida
            c.append(b[0])
            b.pop(0)

        else:
            c.append(a[0])
            a.pop(0)

    # En este punto, alguna de las 2 listas está vacía
    # Se evita hacer operaciones con listas para mostrar mejor el algoritmo

    while len(a) != 0:
        c.append(a[0])
        a.pop(0)

    while len(b) != 0:
        c.append(b[0])
        b.pop(0)

    return c

numeros = [10, -1, -5, 100, 5, -0.5, -22, 44, -100, -10, 0.5, 0, -44, 1, 22]
print(ordenar(numeros))
```

Salida:

```
[100, 44, 22, 10, 5, 1, 0.5, 0, -0.5, -1, -5, -10, -22, -44, -100]
```

25.6. Quick sort

25.7. Heap sort

Capítulo 26

Algoritmos de búsqueda

26.1. Búsqueda lineal

26.2. Búsqueda binaria

Capítulo 27

Algoritmos de matrices

Capítulo 28

Implementación de estructuras de datos

Capítulo 29

La librería NumPy