

PRIMER SEMESTRE 2020

Introducción a la programación en Python

Valentina Álvarez Gálvez & Adrián Soto Suárez

Índice general

1. Prólogo	3
1.1. ¿Cómo utilizar este libro?	3
1.2. ¿Cómo hacernos llegar <i>feedback</i> de este libro?	4
2. Introducción	5
2.1. Introducción a los algoritmos	5
2.2. Creando un programa en Python	9
3. Conceptos básicos	11
3.1. Asignación de variables	11
3.2. Tipos de datos	11
3.3. Operadores	13
3.4. Comandos Básicos	14
3.5. Programas básicos	17
4. Control de flujo	19
4.1. If, elif y else	19
4.2. Loops	23
4.2.1. While	23
4.2.2. For	24
4.2.3. Break y Continue	26
5. Funciones y módulos	28
5.1. Funciones	28
5.2. Módulos	30
5.2.1. Main	32
6. Strings	34
6.1. Declaración y manejo de strings	34
6.2. Métodos de strings	37
6.3. El tipo chr y la tabla ASCII	38
7. Listas	40
7.1. Declaración de listas	40
7.2. Funciones relacionadas con listas	41
7.3. Ordenamiento	47
7.4. Valores por referencia	49

8. Diccionarios y Tuplas	52
8.1. Diccionarios	52
8.2. Tuplas	55
9. Clases	58
9.1. Creando una clase	58
9.2. Agregando métodos	60
9.3. Utilizando una clase	61
9.4. Imprimiendo clases	64
9.5. Atributos de instancia y de clase	67
10. Archivos	71
10.1. Ruta de un archivo	71
10.2. Abrir y cerrar un archivo de texto	72
10.3. Leer un archivo de texto	72
10.4. Escribir un archivo de texto	75
11. Recursión	78
11.1. Definir una función recursiva	78
11.2. Iterar de forma recursiva	79
11.3. Permutación de <i>strings</i>	79
11.4. Ordenamiento con funciones recursivas	81
11.4.1. Quicksort	81
11.4.2. Mergesort	81

Capítulo 1

Prólogo

Este libro nació el año 2013 como un apunte para un curso de la Escuela de Ingeniería de la Universidad Católica de Chile. En ese momento uno de nosotros lo escribió porque estaba aprendiendo Python y quería guardar registro de todos los conceptos. Después, cuando el libro llegó a los alumnos se hizo bastante popular, algo que era más o menos inesperado. Por varias generaciones fue bastante útil, sin embargo se fue perdiendo un poco con el tiempo.

Luego, este lenguaje de programación se volvió bastante popular no solamente para la gente dedicada a la computación. De esta forma muchos amigos y conocidos nos pedían apuntes de Python para empezar a aprender o repasar conceptos. Nosotros hacíamos llegar este libro pero notábamos que le faltaba algo: (1) los códigos no eran de mucha calidad, de hecho no nos sentíamos orgullosos de ellos, (2) las explicaciones podían mejorar bastante y (3) había que mejorar el código fuente de este libro (hecho en Latex, una herramienta muy útil para escribir este tipo de cosas). Esta es una versión *beta* del libro. Aquí vas a encontrar una versión mejorada respecto a la versión original. Ahora nos sentimos realmente orgullosos de lo que hay aquí. Pero en el futuro podrás encontrar:

- Ejercicios propuestos para muchos capítulos de este libro.
- Soluciones en línea que vas a poder correr desde Google Colab, una herramienta de Google para ejecutar código Python en línea.
- En el mediano plazo, este libro será *open source* y vamos a aceptar *pull requests* para mejorarlo.

Este libro está lleno de ejemplos que te servirán para aprender, además de tener referencia a muchos de los conceptos más utilizados del lenguaje. Sin embargo, creemos que este libro no va a reemplazar las asistencias a clases. De la misma forma, este es un punto de partida: para aprender a programar bien esto debe ir acompañado de ejercitación por parte del lector. No se puede aprender a programar si no se meten las manos al código.

1.1. ¿Cómo utilizar este libro?

Este libro posee una [Wiki](#)¹ en la que vas a encontrar detalles como la instalación de Python, la ejecución de un programa de Python y la forma de **ejecutar programas de Python en Google Colab**.

Esto último es muy importante, porque en este libro vas a encontrar ejercicios propuestos². Las soluciones a los ejercicios van a estar en nuestro [repositorio de GitHub](#)³. Si lees el artículo de Wiki sobre Google Colab te debería quedar claro cómo importar estas soluciones para ejecutarlas en línea. Sino, puedes ver el código desde el repositorio

¹El link es <https://github.com/alanezz/PythonBookSolutions/wiki/>.

²En esta versión no son muchos, pero en el futuro habrán más.

³El link es <https://github.com/alanezz/PythonBookSolutions/>.

y copiarlo de forma local en tu computador.

La idea es que vayas revisando las soluciones a medida que avances. Recuerda, programar por tu cuenta es súper importante para aprender a programar.

1.2. ¿Cómo hacemos llegar *feedback* de este libro?

Nuestra forma preferida para recibir *feedback* de este libro es que publiques una *issue*⁴ en el repositorio del libro. También puedes enviarle un correo a Adrián, que para contactarlo puedes revisar su *página personal*⁵.

⁴El link es <https://github.com/alanezz/PythonBookSolutions/issues/>.

⁵El link es <https://adriansoto.cl/>.

Capítulo 2

Introducción

Antes de aprender Python debemos familiarizarnos con algunos conceptos previos. Si bien este libro busca enseñar un lenguaje de programación en particular, no debemos olvidar que el concepto de algoritmo trasciende al lenguaje de programación. Nosotros podemos tener dos programas que hacen lo mismo, pero escritos en distintos lenguajes.

En este capítulo comenzaremos describiendo qué es un algoritmo, para luego definir lo que es un lenguaje de programación. Después hablaremos del lenguaje que utilizaremos durante este libro, que es Python, para finalmente hablar de cómo podemos correr un programa en este lenguaje en nuestros computadores.

2.1. Introducción a los algoritmos

Es fundamental entender el concepto de **algoritmo** antes de aprender cualquier lenguaje de programación. Un algoritmo es un conjunto finito de pasos definidos y ordenados que nos permiten realizar una actividad de manera metódica. Un algoritmo no está amarrado a ningún lenguaje de programación en particular, y más aún, probablemente hay algoritmos que ejecutamos “en la vida” que ni si quiera requieren ser programados. Un ejemplo de esto es seguir una receta de cocina:

Ejemplo 2.1.1. Supongamos que queremos cocinar un plato de tallarines. Para hacer esto debemos seguir siempre una secuencia fija de pasos, que sería calentar el agua en una olla, agregar los tallarines, agregar sal, esperar una cantidad fija de tiempo y servir los tallarines. Esto lo podemos ilustrar con el siguiente diagrama:

En este caso estamos siguiendo una secuencia fija de pasos, ya que siempre que cocinemos tallarines lo haremos de la misma forma. Sin embargo, esto podría cambiar. Esto lo veremos en un segundo ejemplo

Ejemplo 2.1.2. Ahora queremos cocinar tallarines con alguna salsa debido a que tengo invitados a mi casa. Aquí vamos a tener dos casos: agregar una salsa de carne si mis invitados no son vegetarianos, o agregar una salsa de pesto si es que lo son. En este caso ejecutaremos acciones de forma condicional, dependiendo de mis invitados. Un diagrama que representa nuestro nuevo algoritmo se puede ver en la siguiente figura:

En este caso el rombo representa una acción condicional. Dependiendo de la respuesta, tenemos dos posibles acciones. Aunque independientemente de lo que escojamos, el resultado será el mismo: servir el plato. Ahora vamos a ver un ejemplo en el que utilizaremos otra acción especial, que es repetir una acción hasta que se cumpla una condición.

Ejemplo 2.1.3. La última vez que asistieron los invitados, a pesar de dejar los tallarines en la olla el tiempo que se indicaba en el envase, estos quedaron crudos. Por lo mismo, en vez de esperar los minutos que se indican, vamos a probar que estén suficientemente cocidos cada dos minutos, y los sacaremos cuando estén listos. El diagrama de la figura 2.3 ilustra nuestro nuevo algoritmo.

En este caso, el rombo azul representa un condicional que si no se cumple, nos devuelve a una acción por la que pasamos anteriormente. Podemos decir que estaremos en un **ciclo** hasta que se cumpla la condición que nos saca de él.

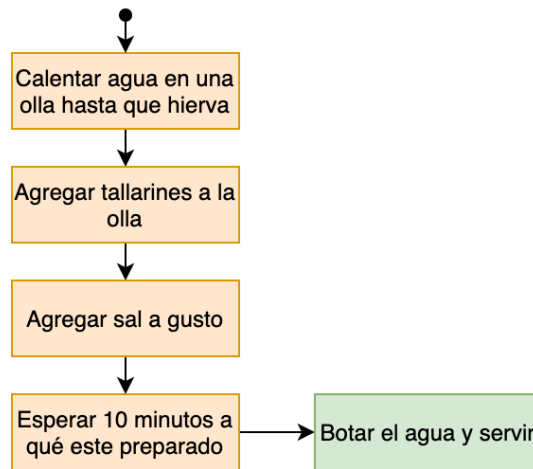


Figura 2.1: algoritmo para cocinar tallarines.

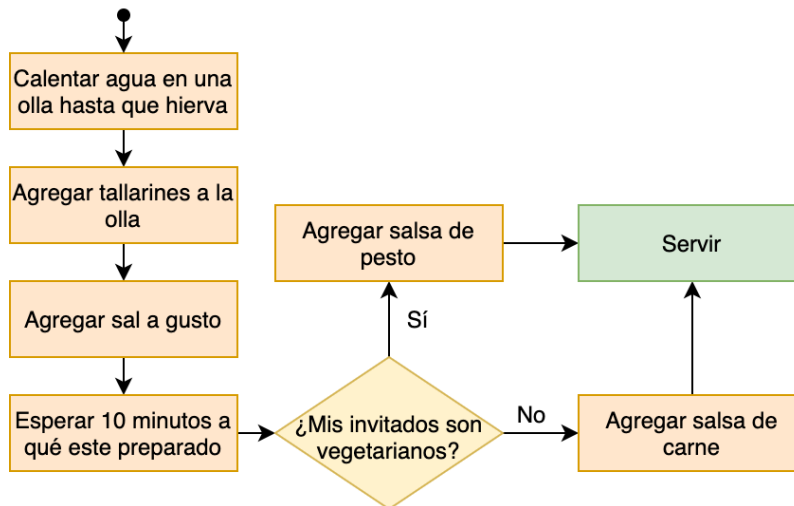


Figura 2.2: algoritmo para cocinar tallarines con alguna salsa, dependiendo de la preferencia de mis invitados.

Del algoritmo al código

Como pudimos apreciar, un algoritmo no necesariamente se vincula a un lenguaje de programación o al hecho de usar un programa. Sin embargo, las aplicaciones que ocupamos día a día se basan en esta idea. Por ejemplo, al intentar entrar a una red social en la que necesitamos iniciar sesión, no se nos mostrarán ciertas páginas si no tenemos la sesión iniciada: esto es un ejemplo de condicionales. A su vez, hay páginas en las que podemos equivocarnos un número determinado de veces. Por ejemplo, si ingresamos mal 3 veces la contraseña en la página de un banco, probablemente seremos bloqueados. Esto es un ejemplo de ciclo: llevamos un contador del número de intentos, y si los sobrepasamos ya no podremos ingresar. Si nos quedan intentos, volveremos a un paso anterior, que es donde la página nos pregunta por nuestra contraseña.

La idea de qué es un algoritmo la podemos seguir complejizando, sin embargo nos vamos a detener ahora ya que este conocimiento se va adquiriendo a medida que uno entiende qué se puede hacer con un lenguaje de programación. En general, las cosas que se pueden hacer en un lenguaje u otro tienden a ser muy similares, y cuando

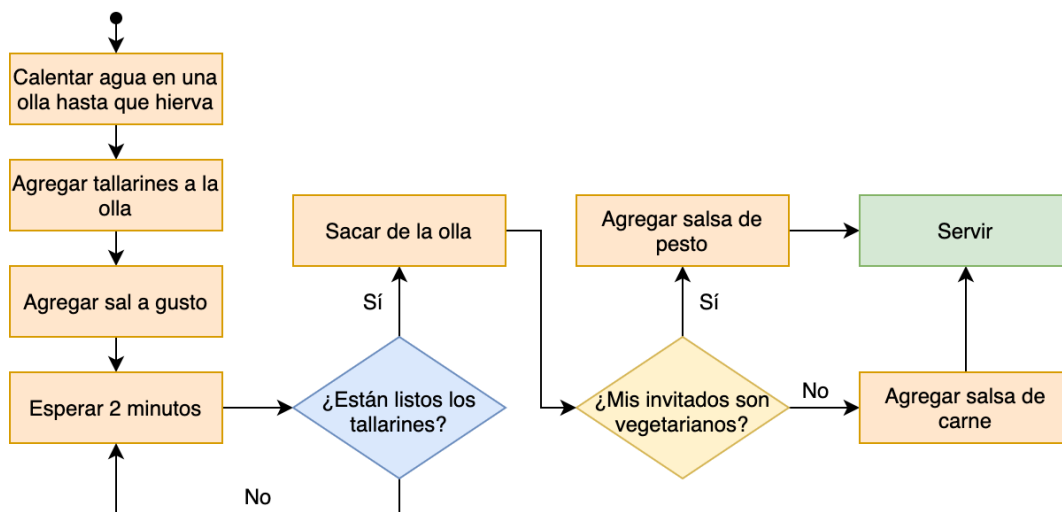


Figura 2.3: algoritmo para cocinar tallarines con alguna salsa, dependiendo de la preferencia de mis invitados.

estamos en niveles básicos de programación, tienden a ser las mismas. Lo único que cambia es la forma en la que se ve el código. Por lo mismo, más que aprender un lenguaje en particular, es buena idea saber modelar el problema y diseñar un algoritmo que lo resuelva, para que después escribir el código en un lenguaje determinado sea inmediato.

Ahora vamos a ver un ejemplo más de algoritmo. Este sí está relacionado a algo que vamos a programar, y de hecho, comúnmente se desarrolla cuando uno está aprendiendo, independiente del lenguaje de programación.

Ejemplo 2.1.4. Vamos a diseñar un algoritmo que dado un número natural positivo n , retorne la suma de los n primeros números naturales. Si bien existe una fórmula que hace esto de forma inmediato, lo vamos a modelar con un ciclo. El algoritmo se resume en el diagrama 2.1.4.

Primero le pedimos al usuario un número n , que es el número hasta el que vamos a sumar. Después vamos a hacer uso de **variables**. Estamos creando dos variables: `suma` y `c`, ambas inicialmente iguales a 0. Luego voy a iterar por el ciclo exactamente n veces, ya que el contador va a aumentar en 1 cada iteración y cuando este sobrepase el valor de n , vamos a salir del ciclo. En cada iteración agregamos el valor de `c` a `suma`. Así, en la primera iteración `suma` valdrá 1, luego 3, luego 6, y así sucesivamente. Finalmente, le mostramos al usuario el resultado.

Ahora vamos a escribir un programa que hace esto en Python. No es la idea entender el código al 100%, pero al menos se debería entender intuitivamente los elementos que conforman al lenguaje.

```

1 n = input('Ingresa el número: ')
2 c = 0
3 suma = 0
4 while c != n
5     c = c + 1
6     suma = suma + c
7 print(suma)

```

Notamos el uso de la instrucción `input`, que pide un número al usuario y lo guarda en la variable `n`. Luego se inicializan las variables `c` y `suma` iguales a 0. Mientras `c` sea distinto de `n`, vamos a ejecutar las instrucciones del ciclo (marcado por la palabra clave `while`) que corresponden a las líneas 5 y 6, que están indentadas. El contador incrementa su valor en 1 y la variable `suma` incrementa su valor según el número en el que el contador `c` está. Después de terminar el ciclo, con la instrucción `print` se entrega el resultado al usuario.

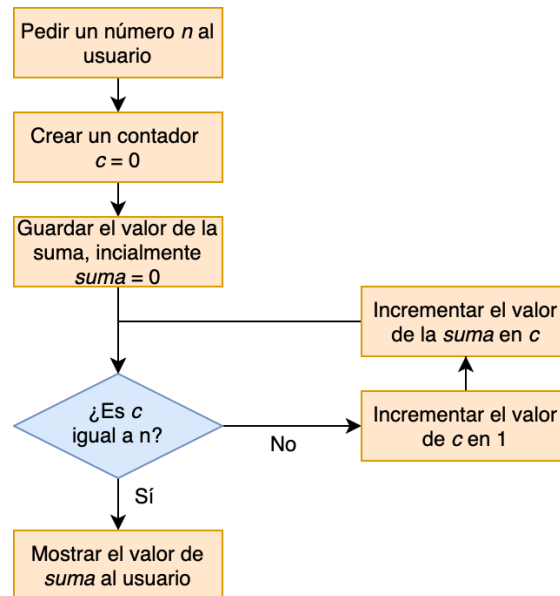


Figura 2.4: algoritmo para retornar la suma de los primeros n naturales.

Notemos que nuestra solución hace uso de un ciclo, pero no es la única forma de resolver esto. Como bien sabemos, la fórmula $\frac{n(n+1)}{2}$ puede calcular el valor, y esto lo podríamos haber escrito con menos líneas de código. En general, para resolver un problema existirán muchas técnicas, unas más fáciles de programar que otras, pero también habrán ciertas soluciones más eficientes que otras. Analizar cuan eficiente es un algoritmo no es algo que vayamos a considerar durante este libro, pero a medida que uno aprende, es ideal ir diseñando soluciones que sean mejores. En general, una modelación buena va a generar un código más eficiente y de mayor calidad (en términos de legibilidad para terceros, por ejemplo).

Algoritmos y lenguajes de programación

Para el problema que acabamos de describir, podríamos hacer un programa en otro lenguaje de programación que lo resuelva. En general, uno utiliza cierto lenguaje dependiendo de lo que se quiere hacer. Por ejemplo, para hacer programas relacionado a computos científicos e inteligencia artificial uno buscaría usar Python. Sin embargo, hace más sentido desarrollar ciertos proyectos de desarrollo web en JavaScript. De la misma forma, si quiero programar *drivers* o deseo hacer código de bajo nivel, voy a tender a usar lenguajes como C.

Es buena idea aprender a programar en Python porque es ampliamente usado en la actualidad, su curva de aprendizaje es rápida y tiene una comunidad muy amplia dispuesta a contestar preguntas. Esto último es muy importante, porque los problemas que vamos a tener a la hora de hacer un programa quizás ya los tuvo alguien más, y siempre es una buena idea buscar en Google antes de entrar en desesperación. En general, muchas respuestas a nuestras preguntas se encuentran fácilmente si sabemos cómo buscar.

Ahora que ya entendemos la noción de algoritmo y vimos nuestro primer código en Python, vamos a descubrir cómo escribir y correr nuestro primer programa en Python.

2.2. Creando un programa en Python

Ahora vamos a hablar de cómo crear un programa en Python y cómo ejecutarlo. Pero antes, es importante que tengas instalado Python en tu computador. En este libro vamos a hablar brevemente de cómo instalar Python y ejecutar un programa en él, sin embargo, en la [Wiki](#) de este libro vas a poder encontrar instrucciones detalladas para cada sistema operativo.

Observación 2.2.1. Durante este libro vamos a asumir que trabajamos con la más reciente versión de Python 3.

Instalación de Python

La instalación de Python depende del sistema operativo. En términos generales las instrucciones son las siguientes:

- **Windows:** puedes descargar Python desde [esta página](#)¹. Después de instalarlo, te recomendamos agregar Python a tus variables de entorno. Para más detalles puedes *googlear* esto o revisar la Wiki.
- **OSX:** Python 3 debería venir instalado en las últimas versiones de OSX, no obstante, si quieres instalarlo por tu cuenta, te recomendamos hacerlo con **Homebrew**, que es un administrador de paquetes para OSX. Primero tienes que instalar Homebrew (puedes buscar cómo hacer esto en Google o en la Wiki del libro) y luego correr el comando `brew install python3`. Una ventaja de esto es que actualizar Python hacia nuevas versiones es sencillo, y además funciona bastante bien con `pip`, el administrador de librerías de Python. Esto es bastante útil a medida que empiezas a aprender otros tópicos. Por ejemplo, probablemente vas a querer instalar la librería **Tensor Flow** si empiezas a desarrollar algoritmos de inteligencia artificial.
- **Linux:** la última versión de Python 3 ya debería estar instalada junto al administrador de librerías.

Ejecutar un programa en Python

Para programar en Python, nosotros recomendamos utilizar un editor de texto; mientras que para ejecutar nuestros programas, recomendamos usar la terminal. Nuestro editor de texto favorito es **Visual Studio Code**, pero también nos gusta **Atom** y **Sublime Text**. En cuanto a la terminal, basta la línea de comandos CMD de Windows², o la terminal por defecto de OSX o Linux.

Para comenzar, vamos a crear un nuevo programa. Para esto, abrimos nuestro editor de texto y creamos un nuevo archivo. En él escribimos lo siguiente:

```
1 print("Hello world!")
```

Y guardamos ese archivo como `programa.py`. Este es nuestro primer programa en Python, que imprimirá en la consola “Hello World”. En general, este programa es el primero que se escribe al aprender un nuevo lenguaje de programación, y si ya programaste anteriormente, esto no debería ser nuevo para ti.

Una vez que guardemos este archivo, debemos abrir una terminal desde la carpeta en la que dejamos el archivo. Una vez ahí ejecutamos el comando:

```
python3 programa.py
```

que ejecuta el programa llamado `programa.py` con Python 3. Lo que deberías ver en la consola como output al programa es:

¹El link es <https://www.python.org/downloads/>.

²Para ocuparla sin problemas tienes que haber agregado Python a las variables de entorno.

```
>>>
Hello world!
>>>
```

Aquí estamos asumiendo familiaridad con el uso de la consola (o terminal). Si no estás familiarizado con esto puedes revisar la [Wiki](#). Finalmente recuerda que hay otras formas de ejecutar un programa en Python, como por ejemplo, usar el editor por defecto de Python llamado **IDLE**, o alguna otra IDE para programar, como **Eclipse** o **PyCharm**. Si estás familiarizado con Java, quizás se te haga natural programar en Python usando Eclipse (tienes que instalar el plugin). En general, si quieres un entorno más robusto puedes probar con PyCharm y ver si te acomoda. El IDLE no lo recomendamos, pues como editor de texto es bastante precario.

También existe una herramienta llamada Anaconda, para instalar Python junto a varias librerías bastante utilizadas. Nosotros **NO** recomendamos el uso de Anaconda, sobre todo si estás familiarizado con ambientes de programación y el uso de terminal.

Google Colab

Es posible ejecutar programas en Python utilizando **Google Colab**, una herramienta de Google del estilo de Google Docs en la que puedes programar en Python a través de *notebooks*. Un *notebook* es un archivo que permite mezclar texto con código en Python. Puedes crear estos *notebooks* en tu computador instalando la librería `jupyter`, pero aquello escapa a los contenidos de este libro.

En la [Wiki](#) puedes encontrar en detalle cómo ejecutar programas escritos en Python en Google Colab. Es importante que lo tengas presente, pues las soluciones a los ejercicios están publicadas en el [repositorio](#) de este libro como *notebooks* que puedes importarlos a Google Colab y ejecutarlos en línea. Todo esto sin tener que instalar nada en tu computador.

Capítulo 3

Conceptos básicos

En este capítulo revisaremos los principales conceptos que debemos conocer antes de empezar a programar en Python. Hablaremos sobre variables, sus tipos, como podemos operar con ellas, algunos comandos necesarios para hacer programas básicos y ejemplos que nos permitirán aplicar lo aprendido hasta el momento.

3.1. Asignación de variables

En computación, una variable es un espacio en memoria reservado para almacenar información que puede o no ser conocida, les daremos un nombre y les asignaremos un valor. Es útil mantener la buena práctica de utilizar nombres descriptivos al momento de nombrar tus variables, esto te ayudará a mantener orden en tu código y poder entenderlo mejor cuando necesites leerlo de nuevo.

Para declarar y asignarle un valor a una variable en Python debemos usar el símbolo igual (=).

```
1 var = 1 # Aquí estamos asignando a la variable con nombre var el valor 1
2 # Notemos que al usar el símbolo # estamos comentando el código.
```

Observación 3.1.1. Hay que tener en consideración que Python diferencia entre mayúsculas y minúsculas (lo que se conoce como *case sensitive*), por lo tanto, no es lo mismo una variable que se llame **var** a otra que se llame **Var**.

3.2. Tipos de datos

En Python, todo elemento tiene un tipo. Este tipo determina las acciones que podemos realizar sobre el elemento o bien el dominio de los valores posibles que puede tomar. Entre los tipos de valores posibles destacan los siguientes:

- **Integer:** una variable de tipo `int` representa un número entero (en inglés *integer*). Puede tomar valores enteros y es posible realizar operaciones aritméticas sobre él. Por ejemplo:

```
1 a = 2 # Asignamos a la variable a el valor 2
2 b = 5 # Asignamos a la variable b el valor 5
3 c = a + b # La variable c tiene el valor 7
4 e = b % a # La variable e tiene el valor 1, que es el resto de 5/2
```

- **Float:** una variable de tipo `float` representa un número racional. Su nombre proviene de *floating point* (punto flotante en español) y es posible realizar operaciones aritméticas sobre él. El siguiente código muestra algunos

procedimientos básicos:

```
1 a = 2.5
2 b = 5.1
3 c = a + b # La variable c tiene el valor 7.6
4 d = 5 / 2 # La variable d tiene el valor 2.5
5 e = 5 // 2 # La variable e tiene el valor 2, que es la división entera entre 5 y 2
6 f = 2 ** 4 # La variable f tiene el valor 16, que es 2 elevado a 4.
```

- **String:** los objetos de tipo `str` son los tipos llamados *strings* (cadena, en español) que sirven para representar texto mediante una “cadena de caracteres”. Es importante notar que la concatenación de dos strings se hace con el operador `+` y que todo texto entre comillas simples representará strings, como por ejemplo:

```
1 texto1 = 'hello'
2 texto2 = 'world'
3 texto3 = texto1 + texto2 # Esta variable tiene valor 'helloworld'
```

Observación 3.2.1. Python es indiferente al uso de comillas simples (‘’) o comillas dobles (‘’) para los *strings*, pero esto no significa que puedas mezclarlos, debes ser consistente con la notación que escojas.

- **Boolean:** los objetos de tipo `bool` son variables que pueden tomar sólo dos valores: `True` o `False`. Son de mucha importancia, pues pueden representar condiciones de término o de fin de un programa o una parte de éste. Una forma de obtener valores booleanos es a partir de comparaciones, las cuales pueden ser las siguientes:
 - Variable *a* es **igual** a variable *b*: `a == b`.
 - Variable *a* es **distinta** a variable *b*: `a != b`.
 - Variable *a* es **mayor o igual** que variable *b*: `a >= b`.
 - Variable *a* es **menor o igual** que variable *b*: `a <= b`.
 - Variable *a* es **mayor** que variable *b*: `a > b`.
 - Variable *a* es **menor** que variable *b*: `a < b`.

Las comparaciones son válidas para los tipos `int`, `float`, `str`, entre otros.

Ejemplos de variables booleanas serían los siguientes:

```
1 var1 = True # La variable var1 tiene valor True
2 var2 = 3 < 5 # La variable var2 tiene valor True, ya que 3 es menor que 5
3 var3 = 2 > 5 # La variable var3 tiene valor False, ya que 2 no es mayor que 5
4 var4 = 5 == 5 # La variable var4 tiene valor True ya que 5 es igual a 5
5 var5 = 5 >= 4 # La variable var5 es True pues 5 es mayor o igual a 4
6 var6 = 5 != 4 # La variable var6 es True pues 5 es distinto de 4
7 var7 = 5 == '5' # La variable var7 es False pues el entero 5 es distinto al string 5
8 text = 'hola'
9 var8 = text == 'hola' # La variable var8 es True
```

Notemos, a partir de los ejemplos anteriores, la diferencia entre (`=`) que se usa para asignación de variables y (`==`) que se usa para comparaciones de igualdad. Esto se visualiza muy bien en las líneas 4, 7 y 9 del código, donde se comparan valores a través de (`==`) y luego el resultado de la comparación respectiva se asigna a las variables `var4`, `var7` y `var8` con (`=`).

- **None**: los objetos de tipo `None` representan variables que no han sido inicializadas, es decir, variables que queremos crear, pero a las que no queremos darle un valor en particular aún.

```
1 var = None # Esta variable tiene valor None
```

Observación 3.2.2. Dado que `None` es un tipo de dato como los mencionados anteriormente, podemos también hacer comparaciones con este tipo.

```
1 var = None
2 comparacion = var == None # La variable comparacion es True
```

3.3. Operadores

Un operador es un símbolo matemático que ejecuta una determinada operación sobre ciertos operandos. Muchos de ellos probablemente los conozcas con anterioridad, pero no necesariamente con la notación particular utilizada en Python. En esta sección podrás encontrar los principales operadores utilizados en este lenguaje.

Operadores Matemáticos

- **a + b**: Si *a* y *b* son de tipo `int` o `float`, entrega la **suma** de ambos. Si *a* y *b* son de tipo `str`, entrega la **concatenación** de ambos.
- **a - b**: Si *a* y *b* son de tipo `int` o `float`, entrega la **resta** de ambos.
- **a * b**: Si *a* y *b* son de tipo `int` o `float`, entrega la **multiplicación** de ambos.
- **a / b**: Si *a* y *b* son de tipo `int` o `float`, entrega la **división** de ambos.
- **a // b**: Si *a* y *b* son de tipo `int` o `float`, entrega la **división entera** de ambos.
- **a % b**: Si *a* y *b* son de tipo `int` o `float`, entrega el **resto** de la división entre ambos.
- **a ** b**: Si *a* y *b* son de tipo `int` o `float`, entrega el resultado de *a* **elevado b**

```
1 a = 10
2 b = 8
3 mult = a * b # Variable mult tiene valor 80
4 div = a / b # Variable div tiene valor 1.25
5 div_ent = a // b # Variable div_ent tiene valor 1
6 resto = a % b # Variable resto tiene valor 2
7 potencia = a ** b # Variable potencia tiene valor 100000000
```

Operadores Booleanos

- **And**: El operador `and` representa al operador lógico \wedge .
- **Or**: El operador `or` representa al operador lógico \vee .
- **Not**: El operador `not` cambia el valor de verdad de una variable booleana.

```

1 # La variable var_not toma valor False pues negamos el valor de 5!=4
2 var_not = not(5 != 4)
3 # La variable var_and1 es False pues and exige que todas las cláusulas sean True
4 var_and1 = (5 > 4) and (10 < 2)
5 # La variable var_and2 es True pues ambas cláusulas son True
6 var_and2 = (2 >= 1) and (3 == 3)
7 # La variable var_or1 es True pues or exige que una de las cláusulas sea True
8 var_or1 = (5 > 4) or (10 < 2)

```

3.4. Comandos Básicos

Python tiene implementadas algunas funciones básicas que son muy importantes para la mayoría de los programas que crearemos en el futuro, especialmente los primeros. Cabe destacar que el concepto de función es bastante amplio y lo veremos con profundidad en los siguientes capítulos, pero por ahora nos bastará con saber que una función recibe ciertos parámetros, ejecuta acciones sobre ellos y puede terminar con el retorno de un valor. Las funciones más básicas (y que necesitamos hasta ahora) son las siguientes:

- **Print:** la función `print` recibe como parámetro algo que queremos mostrar en consola y luego lo imprime.

```

1 print('Hola')
2 var = 2
3 print(var)

```

```

>>>
Hola
2
>>>

```

También podemos imprimir más de un elemento a la vez, incluso si son de diferente tipo, solo debemos pasarle a la función los diferentes parámetros que queremos mostrar, separados por comas. Ten en cuenta que de esta forma los parámetros se imprimirán en una misma línea, separados por un espacio.

```

1 text1 = "Hello"
2 text2 = "world"
3 var_num = 1
4 print(text1, text2, var_num)

```

```

>>>
Hello world 1
>>>

```

Pero a veces puede pasar que queramos imprimir algo un poco más personalizado, quizás no queremos espacios entre las palabras o quizás queremos más de uno. Para esto podemos usar la concatenación de *strings*, teniendo cuidado en pasar a *strings* los elementos que no lo sean, utilizando `str()`.

```

1 nombre = 'Constanza'
2 print('Mi nombre es: ' + nombre)

```

```

>>>
Mi nombre es: Constanza
>>>

```

Veamos un ejemplo que genera `TypeError`, ya que intentamos concatenar un `str` con un `int`.

```
1 num = 2
2 print('El número es: ' + num)

>>>
TypeError: can only concatenate str (not "int") to str
>>>
```

En este caso debemos transformar la variable que es `int` a un `string`.

```
1 num = 2
2 print('El número es: ' + str(num))

>>>
El número es: 2
>>>
```

También puede ser útil saber que por defecto en Python, el fin de línea de `print` es un salto de línea, es por esto que si escribimos varios `print` consecutivos, estos estarán en diferentes líneas.

```
1 print('Línea 1')
2 print('Línea 2')
3 print('Línea 3')

>>>
Línea 1
Línea 2
Línea 3
>>>
```

Si por alguna razón quisiéramos cambiar esta opción que viene por defecto, podemos cambiar el valor del parámetro `end` de la función. Por ejemplo, si queremos que todo quede en una misma línea, queremos que haya un espacio en blanco luego de cada `print`.

```
1 print('Línea 1', end=' ')
2 print('Línea 2', end=' ')
3 print('Línea 3', end=' ')

>>>
Línea 1 Línea 2 Línea 3
>>>
```

Podemos poner cualquier `string` como valor de este parámetro.

```
1 print('Línea 1', end='-')
2 print('Línea 2', end='-')
3 print('Línea 3', end='-')

>>>
Línea 1-Línea 2-Línea 3-
>>>
```


- **Input:** la función `input` recibe como parámetro un *string* y pide además otro por consola, este último puede ser asignado a una variable en el programa.

```
1 nombre = input('Diga un nombre: ')
2 print('El nombre ingresado es: ' + nombre)
```

El programa se queda esperando a que ingresemos algo en consola.

```
>>>
Diga un nombre:
>>>
```

Si escribimos algo y presionamos *enter* el programa continuará su ejecución y lo que hayamos ingresado quedará guardado en la variable `nombre`.

```
>>>
Diga un nombre: Francisca
El nombre ingresado es: Francisca
>>>
```

Notemos que en el ejemplo anterior estábamos trabajando con *strings*, pero en el caso de que quisiéramos pedir números, por ejemplo, y luego trabajar con ellos mediante operadores matemáticos, debemos transformar el *string* ingresado por consola en número mediante `int()` o `float()`. Se debe realizar lo mismo para trabajar con booleanos con `bool()`. A continuación se muestran algunos ejemplos de esto:

```
1 nombre = input('Diga un nombre: ')
2 num1 = int(input ('Ingrese un número: '))
3 num2 = int(input ('Ingrese otro número: '))
4 print('La suma de los números de ' + nombre + ' es: ' + str(num1 + num2))
```

```
>>>
Diga un nombre: Francisca
Ingrese un número: 10
Ingrese otro número: 15
La suma de los números de Francisca es: 25
>>>
```

Tomemos ahora un ejemplo donde nos perjudicaría no hacer la correspondiente transformación:

```
1 num1 = (input('Ingrese un número: '))
2 num2 = (input('Ingrese otro número: '))
3 print(num1 + num2)
```

```
>>>
Ingrese un número: 5
Ingrese otro número: 8
58
>>>
```

En este caso ingresamos 5 y 8, pero en vez de obtener la suma, obtuvimos la concatenación de los valores como *string*.

- **Type:** la función `type` nos puede ser de utilidad cuando queramos saber el tipo de una variable. Recibe como

parámetro una variable y retorna su tipo.

```
1 var_int = 5
2 print(type(var_int))
3 var_float = 2.5
4 print(type(var_float))
5 var_str = 'Esto es un string'
6 print(type(var_str))
7 var_bool = True
8 print(type(var_bool))
```

```
>>>
<class 'int'>
<class 'float'>
<class 'str'>
<class 'bool'>
>>>
```

3.5. Programas básicos

Con los conocimientos adquiridos hasta el momento ya es posible hacer programas básicos que interactúen con el usuario: pedir un input y devolver algo en consola. Los siguientes ejemplos muestran de buena forma lo que somos capaces de hacer:

Ejemplo 3.5.1. Escribe un programa que pida al usuario el lado de un cuadrado y que entregue el área, el perímetro y el valor de su diagonal.

Solución. En este problema debemos hacernos cargo de tres cosas:

- Recibir el valor del input y guardarlo como un `float`.
- Hacer las operaciones aritméticas respectivas.
- Imprimir en pantalla los resultados pedidos.

Notamos que para la primera parte necesitamos usar la función `input`, para la segunda parte debemos hacer las operaciones aritméticas pertinentes y almacenarlas en variables y, finalmente, para la tercera parte debemos hacer uso de la función `print`. La solución propuesta es:

```
1 # Primera parte
2 lado = float(input('Ingrese el lado del cuadrado: '))
3 # Segunda parte
4 perimetro = lado * 4
5 area = lado ** 2
6 diagonal = lado * (2 ** (1/2))
7 # Tercera parte
8 print('El perímetro es: ' + str(perimetro))
9 print('El área es: ' + str(area))
10 print('La diagonal es: ' + str(diagonal))
```

Ejemplo 3.5.2. Imagina que estás en un curso que tiene 3 pruebas que ponderan un 20% cada una y un examen que pondera el 40% restante. El curso se aprueba si el promedio no aproximado del curso es mayor o igual que 4.0.

Escribe un programa que reciba las 4 notas e imprima **True** si aprobaste el curso y **False** en caso contrario.

Solución. Para resolver el ejemplo debemos recibir las notas correspondientes y realizar las operaciones aritméticas correspondientes. El valor resultante de aquellas operaciones aritméticas se compara a 4.0 para ser asignada a una variable booleana. Luego esta se imprime en consola.

```
1 # Se piden los valores
2 n1 = float(input('Ingrese la nota de la prueba 1: '))
3 n2 = float(input('Ingrese la nota de la prueba 2: '))
4 n3 = float(input('Ingrese la nota de la prueba 3: '))
5 ex = float(input('Ingrese la nota del examen : '))
6 # Se calcula el promedio
7 promedio = 0.2 * n1 + 0.2 * n2 + 0.2 * n3 + 0.4 * ex
8 # Se realiza la comparación
9 pasaste = promedio >=4.0
10 # Se imprime respuesta en consola
11 print(pasaste)
```

Notemos que el programa anterior es equivalente a escribir el siguiente código:

```
1 n1 = float(input('Ingrese la nota de la prueba 1: '))
2 n2 = float(input('Ingrese la nota de la prueba 2: '))
3 n3 = float(input('Ingrese la nota de la prueba 3: '))
4 ex = float(input('Ingrese la nota del examen : '))
5 print((0.2 * n1 + 0.2 * n2 + 0.2 * n3 + 0.4 * ex) >= 4.0)
```

Capítulo 4

Control de flujo

En este capítulo veremos cómo podemos controlar el flujo de nuestro programa, es decir, determinar el orden de ejecución de las instrucciones en él. Como por ejemplo, ejecutar cierto fragmento de código repetidas veces o dependiendo de si se cumplen o no algunas condiciones.

4.1. If, elif y else

En el ejemplo 3.5.2, se obtenía como resultado una impresión en pantalla de una variable booleana, la que podía ser `True` o `False`. En general, no vamos a querer imprimir en pantalla la variable booleana como tal, sino que probablemente desearemos un mensaje más personalizado que dependa del valor de esa variable. En los lenguajes de programación, por lo general, existe una manera de ejecutar cierto grupo de acciones dependiendo de si una condición se cumple o no, es decir, si una variable booleana asociada a la sentencia es verdadera o falsa. En Python, la sintaxis es la siguiente:

```
1 if <variable y/o expresión booleana>:  
2     # Aquí van las instrucciones que queremos ejecutar si se cumple la condición.  
3     # Estas instrucciones deben estar indentadas.
```

Lo que pasará con el código de arriba, es que se van a ejecutar las acciones declaradas en el `if` si es que se cumple la condición o si es que la variable booleana tiene valor `True`. Notemos que luego de la variable y/o expresión booleana hay dos puntos (`:`) y que las acciones a ejecutar deben estar **indentadas**, es decir, desplazadas 2 espacios respecto del `if` declarado (también puedes indentar con `tab` en vez de 2 espacios, pero debes ser consistente con tu elección a través del programa).

Con lo visto hasta el momento, el ejemplo 3.5.2 puede ser replanteado de la siguiente manera:

```
1 # Se piden los valores  
2 n1 = float(input('Ingrese la nota de la prueba 1: '))  
3 n2 = float(input('Ingrese la nota de la prueba 2: '))  
4 n3 = float(input('Ingrese la nota de la prueba 3: '))  
5 ex = float(input('Ingrese la nota del examen : '))  
6 # Se calcula el promedio  
7 promedio = 0.2 * n1 + 0.2 * n2 + 0.2 * n3 + 0.4 * ex  
8 # Agregamos la sentencia if  
9 if promedio >= 4.0:  
10     print('Felicitaciones, aprobaste el curso!')
```

Este programa imprime "Felicitaciones, aprobaste el curso!" solo si se cumple que la variable promedio es mayor o igual a 4.0.

Aclaremos que cuando decimos que algo está "dentro de un **if**", nos referimos a todo lo que está indentado con respecto a ese **if**. Notemos además, que puede haber un **if** dentro de otro y que las instrucciones de este deben estar indentadas según corresponda.

Ejemplo 4.1.1. Crea un programa que dado un número, determine si es par o no. En concreto, si el número es par, imprime "Es par" y luego imprime el número elevado al cuadrado. Al término del programa, despédete con un "Adiós mundo!" independiente de si el número era par o no.

```
1 # Pedimos el ingreso del número
2 num = int(input('Ingrese el número: '))
3 # var es un bool que nos indica si el resto al dividir el número por 2, es cero
4 var = num % 2 == 0
5 # Utilizamos la sentencia if con la variable anterior (que puede ser True o False)
6 if var:
7     # Aquí ponemos la acciones que queremos ejecutar solo si var es True
8     # Imprimimos que es par
9     print('Es par')
10    # E imprimimos el número elevado al cuadrado
11    print(num ** 2)
12 # Nos despedimos
13 print('Adiós mundo!')
14 # La línea anterior no está indentada por lo que se ejecutará independiente del valor de var
```

Ahora bien, nos interesará también ejecutar acciones cuando no se cumpla una condición, pero si se cumpla otra. Para esto existe **elif**. Se usa de manera similar a la sentencia **if**: se escribe **elif**, luego la condición, después los dos puntos (**:**) y finalmente se indentan las instrucciones que queremos que se ejecuten si se cumple la condición. No debe estar indentado respecto al **if** original y es posible utilizar más de uno.

```
1 if <condición 1>:
2     # Aquí van las instrucciones que queremos ejecutar si se cumple condición 1.
3 elif <condición 2>:
4     # Aquí irán las instrucciones que queremos ejecutar si es que no se cumple condición 1
5     # Pero si se cumple condición 2.
6 elif <condición 3>:
7     # Podemos usar más de un elif
8     # Las instrucciones de aquí se ejecutarán si no cumple ni condición 1 ni 2, pero si la 3
```

También nos interesa manejar de alguna manera cuando no se cumple ninguna de las condiciones anteriores. Para esto existe **else**, que se usa después de un **if** o **elif**, no recibe condición, pero si debe ir acompañada de dos puntos (**:**).

```
1 if <condición 1>:
2     # Aquí van las instrucciones que queremos ejecutar si se cumple condición 1.
3 elif <condición 2>:
4     # Aquí irán las instrucciones que queremos ejecutar si es que no se cumple condición 1
5     # Pero si se cumple condición 2.
6 else:
7     # Aquí irán las instrucciones que queremos ejecutar cuando no se cumpla condición 1 ni 2.
```

Tenemos que tener en consideración lo siguiente:

- Un **if** puede usarse solo, es decir sin un **elif** o **else** que lo siga.

- Un **else** puede ir después de un **if** o un **elif**.
- Un **elif** puede ir después de un **if** o un **elif**, pero nunca después de un **else**.
- Es posible anidar estas sentencias unas dentro de otras, siempre que se respete la indentación. Por ejemplo:

```

1  if <condición 1>:
2      # Código
3  elif <condición 2>:
4      if <condición 3>:
5          # Código
6      elif <condición 4>:
7          # Código
8  else:
9      # Código
10     if <condición 5>:
11         # Código
12     elif <condición 6>:
13         # Código
14     else:
15         # Código

```

Ejemplo 4.1.2. Diseña un programa que pida al usuario su edad e imprima en pantalla si es mayor de edad o no (consideramos que la mayoría de edad se cumple a los 18).

```

1  # Pedimos al usuario que ingrese su edad
2  edad = int(input('Ingresa tu edad: '))
3  # Si edad es mayor o igual a 18, es mayor de edad
4  if edad >= 18:
5      print('Eres mayor de edad')
6  # En cualquier otro caso, no lo es
7  else:
8      print('No eres mayor de edad')

```

Ejemplo 4.1.3. Diseña un programa que reciba dos números a y b y una operación aritmética (representada por un número) entre las cuatro básicas (suma, resta, multiplicación y división). Si no se ingresó una operación válida, indicarlo. Imprimir en consola los resultados.

```

1  a = float(input('Ingresa el primer número: '))
2  b = float(input('Ingresa el segundo número: '))
3  op = input('Ingresa la operación : \n 1) Suma \n 2) Resta \n 3) Mult \n 4) Div\n')
4  if op == '1':
5      suma = a + b
6      print('La suma es: ' + str(suma))
7  elif op == '2':
8      resta = a - b
9      print('La resta es: ' + str(resta))
10 elif op == '3':
11     mult = a * b
12     print('La multiplicación es: ' + str(mult))
13 elif op == '4':
14     div = a / b
15     print('La división es: ' + str(div))
16 else:

```

```

17 print('La operación ingresada no es válida')
18 print('Adiós!')

```

Antiejemplo. Pedir 3 números que representan la medida de cada uno de los lados de un triángulo. El programa deber imprimir si el triángulo es equilátero, isósceles o escaleno.

Un error muy frecuente es escribir el siguiente código:

```

1 a = float(input('Ingrese el primer lado: '))
2 b = float(input('Ingrese el segundo lado: '))
3 c = float(input('Ingrese el tercer lado: '))
4 if a == b and b == c:
5     print('Equilatero')
6 if a == b or b == c or a == c:
7     print('Isósceles')
8 else:
9     print('Escaleno')

```

Lamentablemente este programa no es correcto, ya que si nuestro *input* corresponde al de un triángulo equilátero, como por ejemplo $a = 1$, $b = 1$ y $c = 1$, el programa imprime en pantalla equilátero e isósceles, lo que es incorrecto. Esto sucede porque si se cumple la primera condición, **no se excluye** la ejecución del segundo **if**, ya que eso lo haría un **elif**. Recordemos que **elif** se ejecuta solamente si no se cumplió la condición anterior, en cambio un **if** no depende de la condición anterior. Una solución correcta sería:

```

1 a = float(input('Ingrese el primer lado: '))
2 b = float(input('Ingrese el segundo lado: '))
3 c = float(input('Ingrese el tercer lado: '))
4 if a == b and b == c:
5     print('Equilatero')
6 elif a == b or b == c or a == c: # En vez de if pusimos elif
7     print('Isósceles')
8 else :
9     print('Escaleno')

```

Ejemplo 4.1.4. Pide al usuario 3 números: a , b y c , estos representarán los coeficientes de una ecuación cuadrática. Imprime en pantalla:

- Que no es cuadrática si $a = 0$
- Que tiene dos soluciones reales si $b^2 - 4ac \geq 0$.
- Que tiene una solución real si $b^2 - 4ac = 0$.
- Que no tiene soluciones reales si $b^2 - 4ac \leq 0$.

Al realizar este problema debemos comprobar si es o no es cuadrática. En el caso de que lo sea, debemos hacer un análisis de sus posibles soluciones:

```

1 a = float(input('Ingrese a: '))
2 b = float(input('Ingrese b: '))
3 c = float(input('Ingrese c: '))
4 if a != 0:
5     if b ** 2 - 4 * a * c >= 0:
6         print('Tiene dos soluciones reales')

```

```

7     elif b ** 2 - 4 * a * c == 0:
8         print('Tiene una solución real')
9     elif b ** 2 - 4 * a * c <= 0:
10        print('Tiene soluciones complejas')
11    else :
12        print('No es cuadrática')

```

4.2. Loops

Al momento de resolver un problema, nos interesará repetir la ejecución de ciertas acciones mientras se cumpla una determinada condición. En Python, podemos realizar esto con las sentencias **while** y **for**, las que ejecutarán las instrucciones que tengan indentadas una y otra vez mientras se sigan cumpliendo las condiciones definidas.

4.2.1. While

La sentencia **while** repite las instrucciones indentadas mientras el valor de una expresión o variable booleana sea **True**.

```

1 while <Condición y/o variable booleana>:
2     #Instrucciones que serán ejecutadas mientras la Condición y/o variable booleana sea True

```

Es importante notar que podemos poner un **while** dentro de otro, siempre y cuando respetemos la indentación. También es posible usar las sentencias condicionales **if**, **elif**, **else**. Al momento de usar **while** en un programa se debe ser cuidadoso, puesto que si la expresión booleana nunca llega a ser falsa, el programa puede quedar en un *loop* infinito y no terminar.

Ejemplo 4.2.1. Hacer un programa que imprima en consola los números del 1 al 10.

Para realizar este programa, debemos tener un variable (que llamaremos **contador**) que diga el número de iteración en la que está el *loop* y luego debemos imprimirla en pantalla según corresponda. Cuando el contador supere el 10, debemos salir del *loop*.

```

1 count = 1
2 # Mientras el valor de count sea menor o igual que 10
3 while count <= 10:
4     # Se imprime el valor de count
5     print(count)
6     # Incrementamos el valor de count en 1
7     count = count + 1
8 #Cuando count sea 11, no volverá a entrar en el loop y se termina el programa

```

Notemos la importancia de incrementar el valor de **count** en cada iteración. Si no lo hiciéramos, **count** valdría 1 siempre y la condición se seguiría cumpliendo infinitamente, por lo que el programa quedaría pegado en un *loop* imprimiendo 1.

Otra opción válida para hacer esto, utilizando una variable booleana, es la siguiente:

```

1 count = 1
2 seguir = True
3 while seguir:

```



```

4 print(count)
5 count = count + 1
6 if count == 11:
7     seguir = False

```

Ejemplo 4.2.2. Hacer un programa que pida al usuario un número natural n e imprima en pantalla su factorial.

Para realizar este programa, además de llevar un contador de la iteración, debemos tener una variable que vaya guardando el valor del factorial “acumulado” hasta esa iteración:

```

1 n = int(input('Ingrese el número n: '))
2 count = 1
3 mult = 1
4 while count <= n:
5     mult = mult * count
6     count = count + 1
7 print(mult)

```

4.2.2. For

La sentencia **for** se utiliza para recorrer los elementos de una secuencia, esto puede ser una lista, una tupla, un diccionario, un conjunto, un *string*, entre otros. Si bien muchos de estos elementos se verán más adelante en el libro, es posible entender lo básico para usar **for** como un *loop*. Posee la siguiente sintaxis:

```

1 for <elemento> in <secuencia>:
2     # Grupo de instrucciones a ejecutar
3     # Aquí se puede usar o no el elemento actual de la secuencia

```

Es necesario, antes de seguir, conocer la función **range**. Esta función retorna una secuencia de números, partiendo por defecto desde 0, incrementando por defecto en 1 y terminando en el número especificado.

```

1 # La secuencia a irá desde 0 a (f-1), de uno en uno
2 a = range(<f>)
3 # La secuencia b irá desde i a (f-1), de uno en uno
4 b = range(<i>, <f>)
5 # La secuencia c irá desde i a (f-1), de s en s.
6 c = range(<i>, <f>, <s>)

```

Algunos ejemplos de uso de esta función:

```

1 # La secuencia es: 0,1,2,3,4
2 a = range(5)
3 # La secuencia es: 2,3,4,5
4 b = range(2, 6)
5 # La secuencia es: 2,4,6,8
6 c = range(2, 9, 2)

```

Ahora podemos utilizar la función **range** combinada con la sentencia **for**:

```

1 for i in range(1, 10):
2     # Acciones indentadas

```

Sabemos que `range(1, 10)` es una secuencia con los números del 1 al 9, por lo tanto `i` tomará esos valores. Esto significa que las acciones indentadas en el `for` se ejecutarán 9 veces.

Ejemplo 4.2.3. Imprimir en pantalla los números pares entre 1 y 10 excluyéndolos.

```
1 for i in range(2, 10, 2):
2     print(i)
```

Ejemplo 4.2.4. Imprimir en pantalla “Hola mundo!” 3 veces.

```
1 for i in range(3):
2     print('Hola mundo!')
```

En este caso, no estamos utilizando el valor de `i` en las instrucciones indentadas en el `for`.

Ejemplo 4.2.5. Pedir un número natural n e imprimir en pantalla los números primos entre 1 y n . Recordemos que un número es primo si solamente es divisible por 1 y por sí mismo.

La solución propuesta es la siguiente:

```
1 n = int(input('Ingrese el número n: '))
2 for i in range(2, n+1):
3     # Partimos asumiendo que i es primo
4     es_primo = True
5     for x in range(2, i):
6         if (i % x) == 0:
7             # Si encontramos un x, tal que i sea divisible por él, i no es primo
8             es_primo = False
9     # Si luego de recorrer todos los x posibles, es_primo sigue siendo True, lo imprimimos
10    if es_primo:
11        print(i)
```

Lo que el código está haciendo es:

- Pedimos el ingreso de n
- Recorremos los valores desde 2 hasta n , a través de una variable i .
- Para cada valor de i hacemos lo siguiente:
 - Partimos asumiendo que i es primo
 - Recorremos los valores desde 2 hasta el actual valor de $i - 1$ a través de la variable x , para verificar si i puede ser dividido por alguno de estos valores. Si encontramos un x que cumpla esto, quiere decir que i no es primo, por lo que actualizamos el valor de la variable `es_primo`.
 - Una vez hallamos recorrido todos los x posibles para ese i , imprimimos i solo si es que el valor `es_primo` sigue siendo `True`.

Notemos que el `if` de la línea 10 no está indentado con respecto al `for` de la línea 5, pues necesitamos haber pasado por todos los valores de x posibles para verificar si i es primo o no. Pero este sí está indentado con respecto al `for` de la línea 3, pues queremos que se verifique, para cada número i desde el 2 al n , si es un número primo o no.

4.2.3. Break y Continue

A veces no deseamos declarar variables booleanas adicionales para continuar o detener un *loop*. Para esto existen dos sentencias llamadas **break** y **continue**. Al ejecutar un **break** dentro de un *loop*, este terminará y no ejecutará ninguna de las instrucciones indentadas que estaban debajo del **break**. Por otro lado, **continue** hace que el *loop* pase a la siguiente iteración saltándose todas las líneas dentro del *loop* que estuviesen después del **continue**.

Ejemplo 4.2.6. Escribe un programa que imprima los primeros 10 naturales en pantalla usando **break** y **continue**.

```
1 count = 1
2 while True:
3     print(count)
4     count += 1 # Esto es lo mismo que escribir count = count + 1
5     if count <= 10:
6         continue
7     else:
8         break
```

Notemos que, con un buen manejo de variables booleanas, es posible realizar lo mismo que haría un **break** o un **continue**.

Ejemplo 4.2.7. Diseña un programa que cumpla los siguientes requisitos:

- Se piden dos números a y b y una operación aritmética entre las cuatro básicas, representada como un número.
- Si la operación ingresada no es válida, el programa debe indicarlo.
- Si la operación ingresada es válida, imprime en consola los resultados correspondientes.
- De alguna manera, el usuario debe poder elegir salir del programa.

Este es un ejemplo típico básico de interacción entre un programa y un usuario. En estos casos, debemos añadir un *loop* que nos mantenga en el programa hasta que deseemos salir. Hay varias formas de hacer esto, pero la elegida es la siguiente:

```
1 while True :
2     a = float(input('Ingrese el primer número a: '))
3     b = float (input('Ingrese el segundo número b: '))
4     menu = 'Ingrese la operación: \n 1)Suma \n 2)Resta \n 3)Mult \n 4)Div \n 5)Salir \n'
5     opcion = input(menu)
6     if opcion == '1':
7         suma = a + b
8         print('La suma es: ' + str(suma))
9     elif opcion == '2':
10        resta = a - b
11        print('La resta es: ' + str(resta))
12    elif opcion == '3':
13        mult = a * b
14        print('La multiplicación es: ' + str(mult))
15    elif opcion == '4':
16        div = a / b
17        print('La división es: ' + str(div))
18    elif opcion == '5':
19        print('Adiós!')
```

```
20     break
21 else :
22     print('La operación ingresada no es válida')
```

Capítulo 5

Funciones y módulos

Ya habíamos hablado anteriormente un poco sobre funciones, pero en este capítulo ahondaremos un poco más en ellas. Veremos cómo crear y usar nuestras propias funciones. Además veremos un poco sobre qué son los módulos, cómo se utilizan y cuál es su función en Python.

5.1. Funciones

Una función es una sección del código que ejecuta ciertas acciones en base a parámetros entregados. Luego puede retornar un valor o no retornar nada. Una función por sí sola no ejecuta nada, la idea es más bien llamarla más adelante en el programa para poder utilizarla.

Una ventaja muy importante que proveen las funciones es poder reutilizar código. Si ya has programado un poco, te debes haber dado cuenta que muchas veces comenzamos a repetir código, cambiando algunos detalles que perfectamente podríamos parametrizar. Para esto, escribimos ese código que siempre se repite en una función y luego solo la llamamos cada vez que queremos utilizar ese segmento de código, cambiando los parámetros de entrada. La sintaxis para crear una función es la siguiente:

```
1 def <Nombre función>(parámetro1 , parámetro2 , ... , parámetroN):
2     # Aquí el código indentado
3     # ...
4     return <valor de salida> # Esta línea puede no estar, si no se desea que retorne algo
```

Luego basta con llamar a la función con su nombre en el código para poder utilizarla. Cabe destacar que para usar una función debe ser declarada antes del código en el que se va a utilizar, aunque si una función usa otra función, esto no necesariamente se debe cumplir. Veamos un ejemplo simple de uso:

Definiremos una función que suma dos números e imprima en pantalla el resultado. No queremos que retorne nada.

```
1 def suma(num1, num2):
2     resultado = num1 + num2
3     print(resultado)
```

Si ejecutamos el código anterior, vemos que no pasa nada, pues solo definimos la función, pero no la hemos utilizado. Agregamos entonces un par de líneas donde la llamamos.

```
1 def suma(num1, num2):
2     resultado = num1 + num2
3     print(resultado)
```

```

4
5 suma(1, 2)
6 suma(5, 10)

>>>
3
15
>>>

```

Revisemos ahora un ejemplo donde queremos retornar un valor. En vez de imprimir el valor del resultado dentro de la función, vamos a retornar ese valor.

```

1 def suma(num1, num2):
2     resultado = num1 + num2
3     return resultado

```

Nuevamente, si ejecutamos solo este código, no pasa nada, porque aún no hemos usado la función. A continuación agregaremos al código un par de líneas en donde **asignaremos a variables los valores que retorna la función**.

```

1 def suma(num1, num2):
2     resultado = num1 + num2
3     return resultado
4
5 res1 = suma(4, 6)
6 print(res1)
7 res2 = suma(19, 20)
8 print(res2)
9 res3 = suma(2, 4)

>>>
10
39
>>>

```

Repitamos ahora el ejemplo 4.2.5 hecho anteriormente, pero ahora aplicando funciones.

Ejemplo 5.1.1. Pedir un número natural n e imprimir en pantalla los números primos entre 1 y n .

Se repite el código hecho en el capítulo anterior, pero esta vez, para determinar si un número es primo o no, se aplicará una función:

```

1 # Definimos la función que verifica si un número es primo
2 def es_primo(num):
3     # num es el parámetro de entrada que representa el número que queremos saber si es primo
4     # Es una variable local y no influye fuera de la función
5     primo = True
6     for i in range(2, num):
7         if (num % i) == 0:
8             primo = False
9     # Retornaremos un bool que indique si num es primo o no
10    return primo
11
12 # Hacemos la parte del programa que pide un número al usuario y utiliza la función anterior
13 n = int(input('Ingrese un número: '))

```

```

14 for i in range(2, n):
15     # Llamamos a la función en cada iteración
16     if es_primo(i):
17         print (i)

```

Ejemplo 5.1.2. Crea tres funciones que, dada la medida del lado de un cuadrado, una retorne su perímetro, otra su área y otra el valor de su diagonal.

```

1 def perimetro(lado):
2     return (lado * 4)
3
4 def area(lado):
5     return (lado ** 2)
6
7 def diagonal(lado):
8     return (lado * (2 ** 0.5))
9
10 # Ahora las usaremos en un programa
11 x = float(input('Ingrese la medida del lado: '))
12 pe = perimetro(x)
13 ar = area(x)
14 print(per)
15 print(ar)
16 # Aquí no asignamos diagonal(x) a una variable, sino que la imprimimos directamente
17 print(diagonal(x))

```

Cabe destacar que una función retorna un valor si y sólo si tiene **return**. Si la función finaliza con un **print** y no posee **return**, esta función no queremos asignarla a una variable, pues esta sería **None**:

```

1 def una_funcion(s):
2     print(s + ' algún string')
3
4 def otra_funcion(s):
5     a = s + ' algún string'
6     return a
7
8 valor1 = una_funcion('Hola') # Esto imprime Hola Algún string
9 print(valor1) # valor1 es None
10 valor2 = otra_funcion('Hola')
11 print(valor2) # valor 2 es Hola algún string

```

Es necesario recalcar que al llegar al **return**, no se sigue ejecutando ningún otro código dentro de la función, aunque este exista en el código más abajo.

5.2. Módulos

Un módulo en Python es una colección de definiciones y declaraciones que pueden ser importadas desde un programa. En general, cuando creamos un programa y lo guardamos con extensión **.py**, estamos creando un módulo. Python trae varios módulos implementados y entre los más destacados están **random** y **math**. Para importar un módulo, una opción es escribir lo siguiente:

```

1 from <Nombre módulo> import <Nombre de lo que deseamos importar>

```

Por ejemplo, si tenemos un módulo llamado `funciones.py` que tiene la función llamada `suma` que queremos definimos anteriormente, lo importamos y usamos de la siguiente manera:

```
1 # Importamos
2 from funciones import suma
3
4 # Usamos
5 var = suma(1, 3)
```

Observación 5.2.1. En general si el módulo que queremos importar es uno de los que hemos creado nosotros, tanto el programa que lo importa, como el módulo a ser importado, deben estar en la misma carpeta. Hay formas de hacer que el módulo no esté en la misma carpeta, pero no lo veremos en este texto.

También podemos importar de la siguiente manera, que puede ser útil cuando queremos importar muchas funciones y valores de un módulo, porque como vimos en la primera forma, teníamos que nombrar cada una de las definiciones que fuéramos a utilizar.

```
1 import <Nombre módulo>
2
3 # Luego para usar lo que deseamos:
4 <Nombre módulo>.<Nombre de lo que se quiere usar>
```

Con el mismo ejemplo anterior, podríamos hacer lo siguiente:

```
1 # Importamos
2 import funciones
3
4 # Usamos
5 var = funciones.suma(1, 3)
```

Además de esta forma, también podemos utilizar `as` para darle un alias a lo que estamos importando. Esto nos serviría, por ejemplo, si el nombre del módulo que estamos importando es muy largo y no deseamos escribirlo completo cada vez que lo usamos.

```
1 # Importamos
2 import funciones as f
3
4 # Usamos
5 var = f.suma(1, 3)
```

Observación 5.2.2. Podemos utilizar `*` para importar todo lo que contiene el módulo. Pero esto es visto como una mala práctica, ya que genera un código poco legible, pues no se sabe exactamente qué se importa y no se puede encontrar fácilmente desde qué módulo se importó una cierta cosa.

```
1 # Importamos
2 from funciones import *
3
4 # Usamos
5 var = suma(1, 3)
```

Observación 5.2.3. A este proceso de ubicar nuestro código en diferentes módulos se le llama **modularizar**. Más adelante veremos que esto nos será de gran utilidad cuando tengamos programas más grandes, pues con esto

distribuimos todo el código en varias partes, lo que hace que cada uno de los módulos sea más simple de leer. Con esto, será más fácil entender el código y encontrar/resolver problemas.

Veamos ahora un ejemplo utilizando el módulo `math` de Python.

Ejemplo 5.2.1. Usando el módulo `math`, dado cierto radio, calcular el área y el perímetro de un círculo.

Vamos a utilizar funciones para resolver este problema. Usaremos la constante `pi` implementada en el módulo `math`:

```
1 # Importamos
2 from math import pi
3
4 # Definimos las funciones
5 def perimetro(radio):
6     return 2 * pi * radio
7
8 def area(radio):
9     return pi * (radio ** 2)
10
11 # Escribimos el programa que pide el input y utiliza las funciones
12 r = float(input('Ingrese el radio: '))
13 print(perimetro(r))
14 print(area(r))
```

5.2.1. Main

Supongamos que tenemos un módulo llamado `funciones` que contiene lo siguiente:

```
1 def suma(a, b):
2     return a + b
3
4 def resta(a, b):
5     return a - b
6
7 def multiplicacion(a, b):
8     return a * b
9
10 print('Este es el módulo de funciones!!!')
```

Y nuestro programa, donde queremos utilizar estas funciones es el siguiente:

```
1 print('Aquí comienza nuestro programa principal!')
2
3 # Importamos
4 import funciones as f
5
6 # Usamos las funciones del módulo
7 s = f.suma(1, 2)
8 r = f.resta(2, 1)
9 m = f.multiplicacion(2, 3)
10
11 # Imprimimos las soluciones
12 print(s)
```

```

13 print(r)
14 print(m)
15
16 print('Aquí termina nuestro programa principal!')

```

Si ejecutamos nuestro programa principal, vemos que también se ejecutó el `print` del módulo `funciones`:

```

1 >>
2 Aquí comienza nuestro programa principal!
3 Este es el módulo de funciones!!
4 3
5 1
6 6
7 Aquí termina nuestro programa principal!
8 >>

```

Esto sucede porque `import`, además de cargar las definiciones del módulo, también ejecuta todas las instrucciones “sueltas” del módulo, como por ejemplo, los `print`. Para evitar este comportamiento, definimos una condición en el módulo que ejecute esas instrucciones que solo cuando ejecutemos específicamente ese módulo y no cuando se está importando. Para eso utilizamos `if __name__ == "__main__"`, que será verdad solo cuando ejecutemos directamente ese módulo. Sin embargo, si lo estamos importando, entonces será falso y por lo tanto, no se ejecutará nada de lo que esté bajo él. Nuestro módulo `funciones` debería quedar así:

```

1 def suma(a, b):
2     return a + b
3
4 def resta(a, b):
5     return a - b
6
7 def multiplicacion(a, b):
8     return a * b
9
10 if __name__ == "__main__":
11     print('Este es el módulo de funciones!!')

```

Ahora, cuando ejecutamos nuestro módulo principal tenemos:

```

1 >>
2 Aquí comienza nuestro programa principal!
3 3
4 1
5 6
6 Aquí termina nuestro programa principal!
7 >>

```

Porque lo escrito bajo `if __name__ == "__main__"` se ejecutará solo si ejecutamos directamente el módulo `funciones`.

```

1 >> python3 funciones.py
2 Este es el módulo de funciones!!
3 >>

```

Observación 5.2.4. Es una buena práctica de programación no escribir código ejecutable en un módulo si sólo queremos definir en él funciones. Si deseamos hacer un programa relativamente grande, el ejecutable debe ir en un módulo separado que importe todos los demás módulos en los que están declaradas las funciones a utilizar.

Capítulo 6

Strings

En capítulos anteriores, habíamos introducido un tipo de dato llamado *string*, que corresponde a la representación de una cadena de caracteres. Este capítulo busca mostrar como manejar strings y cuales son las funciones más importantes, que ya están implementadas en Python, que permiten manipularlos.

6.1. Declaración y manejo de strings

Una variable se declara de tipo *string* cuando se le asignan comillas, ya sean simples (') o dobles (""), como vimos anteriormente. Por ejemplo:

```
1 # Declaramos un string
2 s = 'Hola, soy un string'
3
4 # Recordemos que input guarda como string el texto ingresado por el usuario.
5 s2 = input('Ingrese su texto')
```

También podemos tener un *string* de múltiples líneas:

```
1 s = '''Este es un string
2 que utiliza,
3 más de una línea.
4
5 Incluso tiene saltos de línea'''
6
7 print(s)
```

```
>>>
Este es un string
que utiliza,
más de una línea.
```

```
Incluso tiene saltos de línea
>>>
```

Dado que los *strings* pueden pensarse como arreglos o listas de caracteres (algo que veremos más adelante), es posible manejarlos como tal y utilizar funciones que ya vienen implementadas en Python.

Para saber el **largo de un *string***, podemos usar la función **len**:

```

1 s = 'mensaje'
2 largo = len(s) # Esta variable tiene el valor 7
3 print(largo)

>>>
7
>>>

```

Podemos hacer *indexing*, es decir, acceder al i-ésimo caracter de un *string* de la forma `string[i]`. Por ejemplo:

```

1 s = 'Hola, soy un string'
2 a = s[0] # Esta variable tiene el valor "H"
3 b = s[1] # Esta variable tiene el valor "o"

```

Observación 6.1.1. Notemos que las posiciones en los arreglos van desde 0 en adelante.

Podemos hacer *slicing*, es decir, acceder a más de un caracter de la siguiente manera: `string[i:f]`, que me devuelve el *substring* desde la posición `i` hasta la posición `f-1`. Cuando no se especifica un valor de `i`, se asume que el valor es 0 y si no se especifica un valor de `f`, se asume que es `len` del *string*.

```

1 s = 'Hola, soy un string'
2
3 # Tenemos de la posición 6 a la 8
4 a = s[6:9] # Esta variable tiene el valor "soy"
5
6 # Tenemos desde el inicio hasta la 6
7 b = s[:7] # Esta variable tiene valor "Hola, s"
8
9 # Tenemos desde posición 5 hasta len del string -1, es decir, el final
10 c = s[5:] # Esta variable tiene valor " soy un string"

```

También podemos recorrer un *string* de atrás hacia adelante utilizando posiciones negativas.

```

1 s = 'Hola, soy un string'
2
3 # Posición -1 corresponde a la última posición
4 a = s[-1] # Esta variable tiene el valor "g"
5
6 # Posición -2 corresponde a la penúltima posición
7 b = s[-2] # Esta variable tiene el valor "n"
8
9 # Tenemos desde la posición 1 hasta la -3
10 c = s[1:-2] # Esta variable tiene valor "ola, soy un stri"
11
12 # Tenemos desde la posición -6 hasta la -2
13 d = s[-6:-1] # Esta variable tiene el valor "stri"

```

Para saber si un caracter o un *substring* forma parte de otro *string*, utilizamos `in`, que nos devuelve una variable booleana:

```

1 s = 'Hola, soy un string'
2
3 # var1 toma valor True, pues el caracter a está en s
4 var1 = 'a' in s

```

```

5
6 # var2 toma valor True, pues el substring Hola está en s
7 var2 = 'Hola' in s
8
9 # var3 toma valor False, pues el substring Chao no está en s
10 var3 = 'Chao' in s

```

También podemos comparar *strings* con < o >, según su orden en la tabla ASCII¹.

```

1 s1 = 'Alameda'
2 s2 = 'Providencia'
3 a = s1 < s2 # Esta variable toma valor True
4 b = s1 > s2 # Esta variable toma valor False

```

Hasta ahora hemos usado comillas simples (') para declarar *strings*, pero ¿qué pasa si queremos, por ejemplo, escribir Bernardo O'Higgins?

```

1 nombre = 'Bernardo O'Higgins'
2 print(nombre)

>>>
    nombre = 'Bernardo O'Higgins'
                          ^
SyntaxError: invalid syntax
>>>

```

Esto pasa porque estamos usando las comillas simples para limitar los *strings*, pero además queremos hacer uso de una comilla simple en nuestro texto. Por lo tanto, Python entiende lo siguiente: tenemos 3 comillas simples, las dos primeras (la del comienzo y la que está entre la O y la H) que delimitan el *string* "Bernardo O" y una última que se considera un error de sintaxis porque no está cerrada. Sin embargo, podemos usar comillas simples dentro del *string* de la siguiente forma:

```

1 nombre = 'Bernardo O\'Higgins'
2 print(nombre)

>>>
Bernardo O'Higgins
>>>

```

Lo que hicimos fue anteponer un *backslash* (\) a la comilla del apellido. A esto se le llama **escapar** un caracter y en este caso nos sirve para ubicar en el *string* un caracter que, con lo que sabíamos hasta ahora, no podríamos haber ubicado.

Como vimos en algún momento, podemos utilizar comillas dobles para declarar *strings*. Si decidiéramos declararlos de esta manera, no podríamos usar una comilla doble en medio del *string*.

```

1 texto1 = "Aquí hay una "palabra" entre comillas" # Esto genera error de sintaxis
2 texto2 = "Aquí hay una \"palabra\" entre comillas" # Escapamos las comillas

```

También hay algunos caracteres que si los escapamos, significan algo más. Los más usados son:

- \n: es un salto de línea
- \t: es un *tab*

¹Esta tabla se explica en secciones posteriores.

6.2. Métodos de strings

Existen en Python ciertos métodos o funciones que nos permiten realizar varias acciones muy útiles con los datos de tipo *string*. Estas funciones requieren de una instancia de un *string* para ser usadas y el resultado puede asignarse a una nueva variable:

A continuación, se presentan algunos de los métodos más utilizados. Puedes encontrar más funciones o más información sobre el uso de estas funciones en la documentación oficial de Python.

- **upper()**: retorna un *string* con todos los caracteres alfabéticos en mayúscula.

```
1 s = 'hola'
2 a = s.upper() # a tiene valor "HOLA"
```

- **lower()**: retorna un *string* con todos los caracteres alfabéticos en minúscula.

```
1 s = 'Hola'
2 a = s.lower() # a es "hola"
```

- **strip()**: retorna un *string*, eliminando los espacios de los lados.

```
1 s = ' Hola tengo espacios '
2 a = s.strip() # a es "Hola tengo espacios"
```

- **replace(s1,s2)**: retorna un *string* donde se ha reemplazado s1 por s2.

```
1 s = 'hola hola chao hola'
2 a = s.replace('chao','hola') # a es "hola hola hola hola"
```

- **isalpha()**: retorna True si el *string* contiene sólo caracteres alfabéticos:

```
1 s1 = 'hola'
2 a = s1.isalpha() # a posee el valor True
3
4 s2 = '123'
5 b = s2.isalpha() # b posee el valor False
```

- **count(s1)**: retorna un *int*, que representa cuántas veces s1 aparece dentro del *string*.

```
1 s = 'hola hola chao hola'
2 a = s.count('a') # a vale 4
3 b = s.count("hola") # b vale 3
4 c = s.count("chao") # c vale 1
```

- **find(s1)**: retorna la posición de la primera aparición de s1 en el *string*. Si no existe retorna -1.

```
1 s1 = 'hola'
2 a = s1.find('a') # a vale 3
3
```

```

4 s2 = 'babarooa'
5 b = s2.find('b') # b vale 0
6 c = s2.find('roa') # c vale 4
7 d = s2.find('c') # c vale -1

```

- **format(variables)**: toma las variables ingresadas y las posiciona dentro de los correspondientes {}.

```

1 nombre = 'Marcelo'
2 edad = 25
3 txt = 'Mi nombre es {} y tengo {} años'
4 # a tiene valor "Mi nombre es Marcelo y tengo 25 años"
5 a = txt.format(nombre, edad)

```

También podemos hacerlo de la siguiente forma:

```

1 nombre = 'Marcelo'
2 edad = 25
3 # a tiene valor "Mi nombre es Marcelo y tengo 25 años"
4 a = 'Mi nombre es {} y tengo {} años'.format(nombre, edad)

```

Alternativamente, hay una manera más abreviada de hacer esto:

```

1 nombre = 'Marcelo'
2 edad = 25
3 # a tiene valor "Mi nombre es Marcelo y tengo 25 años"
4 a = f'Mi nombre es {nombre} y tengo {edad} años'

```

Ejemplo 6.2.1. Diseña un programa que reciba un *string* por parte del usuario y compruebe si este posee sólo caracteres alfabéticos. Luego, si se cumple lo anterior, pregúntele al usuario si desea pasar todo el texto a mayúscula o minúscula.

```

1 txt = input('Ingrese su texto: ')
2 if txt.isalpha():
3     opcion = input('Qué desea hacer? \n 1)Mayúsculas \n 2)Minúsculas\n')
4     if opcion == "1":
5         txt = txt.upper()
6         print(txt)
7     elif opcion == "2":
8         txt = txt.lower()
9         print(txt)

```

Es necesario notar que para “actualizar” el valor de `txt`, debemos asignarle a `txt` el nuevo valor de la manera: `txt = txt.lower()`, lo mismo para `txt = txt.upper()`, ya que estas funciones no cambian el valor de `txt`, sino que retornan un nuevo elemento.

6.3. El tipo `chr` y la tabla ASCII

En Python no existe el tipo `chr` por sí solo, sin embargo es posible trabajar con strings de largo 1, es decir, podemos representar caracteres. Un carácter es la representación de 8-bits² asociado a una tabla llamada ASCII. Esta tabla, para cada número entre 0 y 255 posee una representación en forma de carácter.

²8 bits es un número de 8 dígitos en binario, que forma la unidad byte

En Python, para obtener el número asociado a un caracter usamos la función `ord(char)`. En cambio, para obtener el caracter asociado a un número usamos la función `chr(int)`. Por ejemplo:

```
1 char_65 = chr(65) # Esto vale "A"  
2 ord_A = ord('A') # Esto vale 65
```

Es necesario notar que en la tabla ASCII:

- Las letras en mayúscula de la **A** a la **Z** están entre los números 65 y 90.
- Las letras en minúscula de la **a** a la **z** están entre los números 97 y 122.

Capítulo 7

Listas

En este capítulo, aprenderemos sobre una de las estructuras de datos más utilizadas en Python: las listas. Además, revisaremos sus principales propiedades y ejemplos de cómo y cuándo usarlas.

7.1. Declaración de listas

Una lista es un tipo de estructura de dato de Python que es capaz de almacenar una colección de diferentes tipos de dato. Por ejemplo, sabemos que podemos guardar en una variable un número entero:

```
1 a = 1
```

pero también, podemos guardar en una variable dos o más números enteros. Por ejemplo ahora asignaremos a la variable `a` los enteros 1, 3 y 4:

```
1 a = [1, 3, 4]
```

Aquí estamos declarando una lista, que está “coleccionando” los elementos 1, 3 y 4. Ahora veremos en detalle cómo funciona este nuevo tipo de dato, que nos ofrece muchas posibilidades nuevas.

Observación 7.1.1. Las listas son estructuras dinámicas ya que su largo no es fijo. Bajo este contexto no se debe confundir a la lista con otra estructura de datos de otros lenguajes llamada **arreglo**. Los arreglos tienen un comportamiento similar a las listas, pero su largo es fijo después de su creación, sin embargo, es posible modificar lo que tenga el término i -ésimo del arreglo.

Listas Unidimensionales

Podemos declarar listas unidimensionales de la siguiente forma:

```
1 # Lista vacía
2 lista_vacia = []
3 # Lista con los elementos 1,2 y 3
4 mi_lista = [1,2,3]
```

Como vemos, `mi_lista` es una lista unidimensional, ya que vamos guardando elementos de tipo `int` uno tras de otro. Cabe destacar que, a diferencia de otros lenguajes, es posible tener elementos de diferente tipo en una lista, por ejemplo, `int`, `float`, `str`, entre otros.

Listas bidimensionales

Las listas bidimensionales, por otro lado, son listas de listas. Un ejemplo sería el siguiente:

```
1 # Lista bidimensional
2 lista_bidim = [[1,2], [3,4]]
```

Notemos que es posible interpretar las listas bidimensionales como matrices, ya que cada una de las listas interiores podría corresponder a una fila de la matriz, y si todas estas listas interiores tienen el mismo largo, este largo corresponde al número de columna. También podemos interpretarlo al revés, es decir, que cada lista interior corresponde a una columna, pero esto queda a criterio de cada uno, siempre y cuando se mantenga la consistencia.

Recorriendo una lista

Como vimos en el capítulo 4, podemos utilizar la sentencia **for** para recorrer secuencias, ahora vemos que las podemos utilizar para recorrer listas.

```
1 for <elemento> in <lista>:
2     # Grupo de instrucciones a ejecutar
3     # Pueden usar o no al elemento
```

Un ejemplo de esto:

```
1 lista = ['palabra1', 'palabra2', 'palabra3']
2 for elemento in lista:
3     print(elemento)
```

```
>>>
palabra1
palabra2
palabra3
>>>
```

Notemos que en el código anterior, nosotros elegimos qué variable usar para designar el elemento de la iteración. Particularmente, en este caso hemos elegido la variable `elemento`.

En vez de recorrer cada elemento de la lista, podemos utilizar las posiciones para recorrerla y entonces, en cada iteración, seleccionar el elemento de la iteración correspondiente. De esta manera, el código anterior es equivalente a:

```
1 lista = ['palabra1', 'palabra2', 'palabra3']
2 for i in range(len(lista)):
3     print(lista[i])
```

Recordemos que cuando estábamos trabajando con *string*, utilizábamos la función `len()` para saber su largo. Ahora, con listas, podemos hacer lo mismo.

7.2. Funciones relacionadas con listas

Python tiene implementados diferentes métodos o funciones que nos pueden ser de mucha utilidad a la hora de trabajar con listas.

Métodos de listas

- **append(e)**: este método nos permite agregar el elemento e a la lista.

```
1 # Creamos una lista vacía
2 mi_lista = []
3
4 # Agregamos elementos de diferentes tipos
5 mi_lista.append(1)
6 mi_lista.append(2)
7 mi_lista.append('elemento')
8 mi_lista.append(True)
9
10 # Imprimimos
11 print(mi_lista)

>>>
[1, 2, 'elemento', True]
>>>
```

- **pop()**: este método nos permite eliminar el último elemento de la lista, que además, puede ser asignado a una variable

```
1 # Tenemos la siguiente lista
2 mi_lista = [0, 1, 2, 3, 4, 5]
3
4 # Utilizamos pop, asignandolo a una variable. Eliminamos el 5
5 elemento_eliminado = mi_lista.pop()
6
7 # Pero también podemos no asignarlo. Eliminamos el 4
8 mi_lista.pop()
9
10 # Imprimimos
11 print(mi_lista)
12 print(elemento_eliminado)

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

Con esta función, también podemos eliminar un elemento según su posición. Para esto, le pasamos como parámetro la posición del elemento que queremos eliminar.

```
1 # Tenemos la siguiente lista
2 mi_lista = [0, 1, 2, 3, 4, 5]
3
4 # Lo asignamos a una variable. Eliminamos elemento en posición 1
5 # La lista quedaría [0, 2, 3, 4, 5]
6 elemento_eliminado = mi_lista.pop(1)
7
8 # No lo asignamos. Eliminamos elemento en posición 0
9 # La lista quedaría [2, 3, 4, 5]
10 mi_lista.pop(0)
11
```

```
12 # Imprimimos
13 print(mi_lista)
14 print(elemento_eliminado)
```

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

- **remove(e)**: elimina el elemento e de la lista. Si e no existe en la lista, obtendremos un `ValueError`.

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

```
>>>
[2, 3, 4]
>>>
```

- **extend()**: este método también nos permite agregar elementos a la lista, pero a diferencia del método `append`, pasamos como argumento algo que podemos recorrer (como por ejemplo otra lista), agregando cada uno de los elementos.

```
1 lista = [1, 2, 3, 4]
2 print(lista)
3 lista2 = [5, 6, 7]
4 lista.extend(lista2)
5 print(lista)
6 lista.extend([8, 9])
7 print(lista)
```

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

- **reverse()**: nos permite invertir los elementos de la lista.

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

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

- **index(e)**: retorna el entero que representa la posición de la primera aparición del elemento e en la lista. Si e no existe en la lista, obtendremos un `ValueError`.

```
1 lista = [1, 2, 3, 4]
2 otra_lista = ['a', 'b', 'c', 'a']
3
```

```

4 pos1 = lista.index(1) # Esto es 0
5 pos2 = lista.index(3) # Esto es 2
6 pos3 = lista.index(5) # Esto genera ValueError porque el 5 no está en la lista
7 pos4 = otra_lista.index('a') # Esto es 0
8 pos5 = otra_lista.index('b') # Esto es 1
9 pos6 = otra_lista.index('d') # Esto genera ValueError porque d no está en la lista

```

- **count(e)**: retorna la cantidad de veces que e se encuentra en la lista.

```

1 lista = [1, 1, 2, 2, 2, 3, 4, 5, 5, 5, 5]
2 a = lista.count(1) # Esto es 2
3 b = lista.count(2) # Esto es 3
4 c = lista.count(3) # Esto es 1
5 d = lista.count(5) # Esto es 4
6 e = lista.count(6) # Esto es 0

```

Otros métodos

- **Indexing**: una vez inicializada la lista, podemos acceder al i-ésimo elemento de la misma manera en que lo hacíamos con los *strings*

```

1 lista = [2, 4, 6, 8, 10, 12]
2
3 # El primer elemento de la lista (2)
4 a = lista[0]
5
6 # El segundo elementos de la lista (4)
7 b = lista[1]
8
9 # Última posición (12)
10 c = lista[-1]
11
12 # Penúltima posición (10)
13 d = lista[-2]

```

Además de acceder al i-ésimo elemento, podemos modificar el i-ésimo elemento, dado que las listas son mutables.

```

1 lista = [1, 2, 3]
2 lista[2] = 4 # Ahora lista es [1, 2, 4]

```

También podemos hacer *indexing* en listas bidimensionales. Sea `lista_bid` una lista bidimensional, puedo acceder al j-ésimo elemento de la i-ésima lista de la siguiente manera: `lista_bid[i][j]`. Veamos algunos ejemplos.

```

1 matriz = [
2     ['a', 'b', 'c', 'd'],
3     ['f', 'g', 'h', 'i'],
4     ['j', 'k', 'l', 'm']
5 ]
6
7 # Esta variable es 'a'
8 var1 = matriz[0][0]

```

```

9
10 # Esta variable es 'c'
11 var2 = matriz[0][2]
12
13 # Esta variable es 'g'
14 var3 = matriz[1][1]
15
16 # Esta variable es 'm'
17 var4 = matriz[2][3]

```

- **Slicing:** una vez inicializada la lista, podemos acceder a partes de la lista, de la misma manera en que lo hacíamos con los *strings*.

```

1 lista = [2, 4, 6, 8, 10, 12]
2
3 # Todos los elementos ([2,4,6,8,10,12])
4 a = lista[:]
5
6 # Desde la posición 2 a la 4 ([6,8,10])
7 b = lista[2:5]
8
9 # Desde el inicio hasta la posición 3 ([2,4,6,8])
10 c = lista[:4]
11
12 # Desde la posición 2 hasta el final ([6,8,10,12])
13 d = lista[2:]
14
15 # Desde la posición 1 a la -3 ([4, 6, 8])
16 e = lista[1:-2]
17
18 # Cambiaremos los valores desde la posición 1 a la 2
19 lista[1:3] = ['a', 'a'] # Ahora es [2, 'a', 'a', 8, 10, 12]

```

Observación 7.2.1. Notemos que al hacer *slicing*, obtenemos una **lista** con los elementos seleccionados.

También podemos hacer *slicing* en listas bidimensionales.

```

1 matriz = [
2     ['a', 'b', 'c', 'd'],
3     ['f', 'g', 'h', 'i'],
4     ['j', 'k', 'l', 'm']
5 ]
6
7 # Obtenemos la primera fila (['a', 'b', 'c', 'd'])
8 var1 = matriz[0][:]
9
10 # Obtenemos los primeros 2 elementos de la tercera fila (['j', 'k'])
11 var2 = matriz[2][:2]

```

- **Unión:** Si tenemos dos listas l1 y l2, podemos unir las listas con el operador +.

```

1 lista1 = [1, 2, 3]
2 lista2 = [4, 5, 6]
3 union = lista1 + lista2 # Esto es [1, 2, 3, 4, 5, 6]

```

- **len(mi_lista)**: nos permite conocer el largo de la lista `mi_lista`, es decir, la cantidad de elementos almacenados en ella.

```
1 mi_lista = [1, 2, 3, 4, 5]
2
3 largo = len(mi_lista) # Esta variable vale 5
```

Métodos con strings

- **string1.split(string2)**: se divide el string `string1` por el separador `string2`. Se retorna una lista y el string original `string1` no se ve modificado. El separador por defecto es un espacio en blanco.

```
1 s1='Soy|un|string|con|rayas'
2 a = s1.split('|') # Esto es ['Soy', 'un', 'string', 'con', 'rayas']
3 s2='Yo no se que soy'
4 b = s2.split('no') # Esto es ['Yo ', ' se que soy']
5 s3 = 'A mi me separan espacios'
6 c = s3.split() # Esto es ['A', 'mi', 'me', 'separan', 'espacios']
```

- **list(string1)**: nos permite transformar un string `string1` a una lista de caracteres.

```
1 string1 = 'Hola mundo!'
2 lista = list(string1) # Esto es ['H','o','l','a', ' ', 'm', 'u', 'n', 'd', 'o', '!']
```

Esto último también funciona para pasar el iterable de `range()` a una lista de números.

```
1 var = range(10)
2 print(var)
3 lista = list(var)
4 print(lista)

>>>
range(0, 10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>
```

Ejemplo 7.2.1. Crea un programa en Python que simule una bitácora. El programa debe pedirle al usuario elegir entre (1) Agregar un nuevo suceso, (2) Leer la bitácora o (3) Salir. Si desea agregar un nuevo suceso, el programa debe además pedir la fecha del suceso.

```
1 # Modelamos la bitacora como una lista
2 bitacora = []
3
4 elegir = True
5 while elegir:
6     opcion = input('Qué deseas hacer? \n 1)Agregar suceso \n 2)Leer bitácora \n 3)Salir \n')
7
8     # Agregamos un suceso
9     if opcion == '1':
10        # Asumimos que el usuario ingresa bien los siguientes datos
11        year = input('Year?: ')
12        month = input('Month?: ')
```

```

13     day = input('Day?: ')
14     suceso = input('Ingrese el suceso: ')
15     # Generamos el string que insertaremos en la bitacora
16     s = year + '/' + month + '/' + day + '\n' + suceso + '\n'
17     bitacora.append(s)
18
19     # Leemos la bitacora
20     elif opcion == '2':
21         for b in bitacora:
22             print(b)
23
24     # Salimos del loop
25     elif opcion == '3':
26         print('Adiós!')
27         elegir = False
28
29     # El usuario ingresa opción no válida
30     else:
31         print('Opción inválida')

```

7.3. Ordenamiento

Sort y Sorted

Python posee una función de ordenamiento llamada `sort()` que es capaz de ordenar **alfabéticamente** (según la tabla ASCII) una lista de *strings* y de **menor a mayor** un grupo de números. También tenemos `sorted()` que retorna una copia de la lista con los elementos ordenados.

Ejemplo de `sort`:

```

1 desordenada = [3, 4, 1, 6, 8, 2, 4]
2 print(desordenada)
3 desordenada.sort() # Aplicamos sort a la lista desordenada
4 print(desordenada)

>>>
[3, 4, 1, 6, 8, 2, 4]
[1, 2, 3, 4, 4, 6, 8]
>>>

```

Ejemplo de `sorted`:

```

1 desordenada = [3, 4, 1, 6, 8, 2, 4]
2 print(desordenada)
3 ordenada = sorted(desordenada) # Debemos asignar el uso de sorted a una nueva variable
4 print(ordenada)

>>>
[3, 4, 1, 6, 8, 2, 4]
[1, 2, 3, 4, 4, 6, 8]
>>>

```


Personalizando el ordenamiento

Podemos personalizar la manera en que el método `sort()` ordena de la siguiente manera:

- **Reverse:** habíamos dicho que `sort()` ordena, por defecto, de menor a mayor los números y alfabéticamente según la tabla ASCII los *strings*. Podemos cambiar a un orden descendente si cambiamos a `True` el valor del argumento `reverse`:

```
1 lista = [3, 1, 4, 2, 5]
2 lista.sort(reverse=True)
3 print(lista)

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

- **Ordenando por otro término:** supongamos que tenemos una lista de personas, donde cada persona es una lista con dos elementos, por ejemplo, nombre y apellido. Si aplicamos `sort()` a `personas`, esta se ordenará según nombre, ya que es el primer elemento:

```
1 personas = [['John', 'Doe'], ['Lucy', 'Cechtelar'], ['Adaline', 'Reichel']]
2 personas.sort()
3 print(personas)

>>>
[['Adaline', 'Reichel'], ['John', 'Doe'], ['Lucy', 'Cechtelar']]
>>>
```

Pero ¿qué pasa si quisiéramos ordenar `personas` según el apellido?. Esto lo podemos hacer de la siguiente forma:

- Definimos una función que recibe como argumento cada elemento de la lista y retorna el término según el cual queremos ordenar. En este caso, cada elemento de la lista `personas` correspondería a una persona en particular. Por otro lado, como queremos ordenar según el apellido, debemos retornar lo que hay en la posición 1 de cada elemento.
- Utilizamos `sort()`, seteando el argumento `key` con el nombre de la función que definimos previamente.

```
1 personas = [['John', 'Doe'], ['Lucy', 'Cechtelar'], ['Adaline', 'Reichel']]
2
3 # Definimos la función que retorna el apellido de la persona
4 def sortApellido(persona):
5     return persona[1]
6
7 # Seteamos key con el nombre de la función anterior
8 personas.sort(key=sortApellido)
9 print(personas)

>>>
[['Lucy', 'Cechtelar'], ['John', 'Doe'], ['Adaline', 'Reichel']]
>>>
```

Si bien Python posee estos métodos de ordenamiento, ¿podemos programar uno nosotros? La respuesta es sí. Existen diversos algoritmos que sirven para ordenar. Detallaremos dos de ellos:

- **Bubble Sort:** Me paro en un término y veo el término siguiente. Si están en orden incorrecto los volteo. Es necesario recorrer desde el principio hasta el final una lista muchas veces para lograr una lista completamente ordenada.
- **Insertion Sort:** Se toma el primer elemento y se fija. Luego se elige el menor de todos los demás y se ubica antes o después (dependiendo si es mayor o menor) del término original. Luego tenemos dos elementos ordenados. Ahora se vuelven a tomar los restantes y se elige el menor y se ordena respecto a los primeros dos (que teníamos fijos y ya estaban ordenados). En general tenemos elementos ordenados y otros que faltan por ordenar. Se toma el menor de los que falta por ordenar y se ubica en su posición correspondiente en los elementos ya ordenados. Se hace esto hasta obtener una lista completamente ordenada.

7.4. Valores por referencia

Supongamos que tenemos el siguiente código:

```
1 a = [1, 2, 3]
2 b = a
```

Luego, si edito un valor en b, este **también se editará en a**, ya que al decir `b = a`, a y b son ahora la misma lista.

```
1 # Situación inicial
2 a = [1, 2, 3]
3 b = a
4 print('a inicial: ', a)
5 print('b inicial: ', b)
6
7 # Modificamos b
8 b.append('otro elemento')
9 print('a después: ', a)
10 print('b después: ', b)
```

```
>>>
a inicial: [1, 2, 3]
b inicial: [1, 2, 3]
a después: [1, 2, 3, 'otro elemento']
b después: [1, 2, 3, 'otro elemento']
>>>
```

Esto sucede porque las listas son tipos por **referencia**, es decir, no nos importan los valores, sino que sus direcciones en memoria. Hasta ahora habíamos trabajado sólo con tipos por valor. Este concepto es muy amplio y se detalla en otros cursos, por ende, ahora sólo debemos quedarnos con que el código anterior es problemático. Una posible solución es:

```
1 a = [1, 2, 3]
2 b = []
3 for i in a:
4     b.append(i)
```

En este caso b es una lista diferente, que no tiene la referencia a la lista a, por lo que una modificación en b no afectará a la original.

```
1 # Situación inicial
```

```

2 a = [1, 2, 3]
3 b = []
4 for i in a:
5     b.append(i)
6 print('a inicial: ', a)
7 print('b inicial: ', b)
8
9 # Modificamos b
10 b.append('otro elemento')
11 print('a después: ', a)
12 print('b después: ', b)

>>>
a inicial: [1, 2, 3]
b inicial: [1, 2, 3]
a después: [1, 2, 3]
b después: [1, 2, 3, 'otro elemento']
>>>

```

Esto también pasa cuando creo una lista bidimensional a partir de la misma lista:

```

1 # Situación inicial
2 lista = ['x', 'x', 'x']
3 matriz = []
4 matriz.append(lista)
5 matriz.append(lista)
6 matriz.append(lista)
7
8 print(matriz)

>>>
[['x', 'x', 'x'], ['x', 'x', 'x'], ['x', 'x', 'x']]
>>>

```

Si edito uno de los tres arreglos interiores, se editarán los tres debido a que `matriz[0][0]` hace referencia al primer elemento de `lista` y dado que este se modifica y que las demás filas tienen la referencia a `lista`, entonces se modifican todas:

```

1 # Situación inicial
2 lista = ['x', 'x', 'x']
3 matriz = []
4 matriz.append(lista)
5 matriz.append(lista)
6 matriz.append(lista)
7
8 # Modificamos
9 matriz[0][0] = "o"
10
11 print(matriz)

>>>
[['o', 'x', 'x'], ['o', 'x', 'x'], ['o', 'x', 'x']]
>>>

```

Para solucionar este problema podemos hacer lo siguiente:

```

1 # Situación inicial
2 matriz = []
3 for i in range(3):
4     lista = []
5     for j in range(3):
6         lista.append('x')
7     matriz.append(lista)
8
9 print('Matriz inicial: ', matriz)
10
11 # Modificamos
12 matriz[0][0] = "o"
13
14 print('Matriz después: ', matriz)

>>>
Matriz inicial: [['x', 'x', 'x'], ['x', 'x', 'x'], ['x', 'x', 'x']]
Matriz después: [['o', 'x', 'x'], ['x', 'x', 'x'], ['x', 'x', 'x']]
>>>

```

Ejemplo 7.4.1. Sea M una matriz de 2×2 . Crea un programa que pida al usuario los elementos de la matriz M . Crea una función que calcule el determinante dada una matriz (representada como lista bidimensional).

Interpretamos la matriz bidimensional como $[[a,b],[c,d]]$ donde el determinante es $ad - bc$.

```

1 # Definimos la función que calcula el determinante
2 def determinante(m):
3     ad = (m[0][0])*(m[1][1])
4     bc = (m[0][1])*(m[1][0])
5     det = ad - bc
6     return det
7
8 # Pedimos al usuario que ingrese los elementos
9 a = float(input('Ingrese a: '))
10 b = float(input('Ingrese b: '))
11 c = float(input('Ingrese c: '))
12 d = float(input('Ingrese d: '))
13
14 # Armamos la matriz
15 primera_fila = [a, b]
16 segunda_fila = [c, d]
17 mi_matriz = [primera_fila, segunda_fila]
18 print(determinante(mi_matriz))

```

Capítulo 8

Diccionarios y Tuplas

En este capítulo aprenderemos sobre otras estructuras de datos muy utilizadas en Python: diccionarios y tuplas. Para cada una de ellas revisaremos cómo se utilizan y en qué casos es conveniente usarlas.

8.1. Diccionarios

Qué es un diccionario

En computación, un diccionario es una estructura de datos que nos permite insertar, eliminar y buscar elementos en él. Se caracterizan principalmente por ser **Mutable**s y **Sin Orden**.

Lo que se almacena en un diccionario es una **llave** (*key*) junto a su **valor** (*value*). Por ejemplo, yo puedo almacenar algo del estilo `{1 : 'mi_palabra'}`, donde la llave es `1` y su valor es `'mi_palabra'`. Si yo le pregunto al diccionario el valor asociado a la llave `1`, este me contestará `'mi_palabra'`, sin embargo, si yo le pregunto por `'mi_palabra'`, el diccionario no sabrá contestarme.

El diccionario que veremos a continuación se conoce habitualmente como una tabla de hash. Si bien su construcción y funcionamiento se explican en cursos superiores, a grandes rasgos su funcionamiento es el siguiente:

- Cuando inicializo un diccionario, reservo un determinado número de buckets (casilleros) en mi memoria. Cada uno tiene un número de bucket asociado.
- Cuando quiero almacenar una llave, ingreso su valor a una función $h(x)$ que me retorna un número entero, el que me indicará el número del bucket en que guardaré el valor.
- Si el bucket estaba ocupado, comienzo a generar una lista con todos los valores que se van acumulando. Mientras más valores acumule en un casillero, más lento se volverá mi diccionario.
- Cuando quiero buscar el elemento por su llave, calculo su función $h(x)$ que me dirá el casillero en el que está. Luego me muevo en la lista respectiva hasta encontrar el valor exacto. Luego lo retorno.
- Cuando quiero eliminar un elemento, realizo la misma acción que en el punto anterior, sólo que en vez de retornar el elemento, lo elimino.

En este capítulo no implementaremos nuestro propio diccionario, sino que aprenderemos a usar el que viene implementado en Python.

Uso de diccionarios

En Python, tenemos dos formas de instanciar un nuevo diccionario. La primera de ellas es usando la función `dict()`, que genera un diccionario vacío:

```
1 mi_diccionario = dict() # Este es un diccionario vacío
```

La segunda forma es a través de llaves:

```
1 mi_diccionario = {} # Este es un diccionario vacío
```

Con esta segunda forma, podemos inmediatamente escribir los *keys* y *values* que queremos que contenga nuestro diccionario:

```
1 mi_diccionario = {1: 'palabra1', 2: 'palabra2'}
```

Independiente de la forma en la que creamos nuestro diccionario, la forma de insertar, buscar y eliminar es la misma:

- Para **buscar** un valor por su respectiva llave *k* debemos hacer lo siguiente:

```
1 valor = diccionario[k]
```

Por ejemplo:

```
1 valor = mi_diccionario[2]
2 print(valor)
```

```
>>>
palabra2
>>>
```

- Para **insertar** en el diccionario una llave *k* junto a su valor *value* debemos hacer lo siguiente:

```
1 diccionario[k] = value
```

Por ejemplo:

```
1 mi_diccionario[3] = 'palabra3'
2 print(mi_diccionario)
```

```
>>>
{1: 'palabra1', 2: 'palabra2', 3: 'palabra3'}
>>>
```

- Para **eliminar** la llave *k* con su respectivo valor asociado debemos hacer lo siguiente:

```
1 del diccionario[key]
```

Por ejemplo:

```
1 del mi_diccionario[2]
2 print(mi_diccionario)
```

```
>>>
{1: 'palabra1', 3: 'palabra3'}
>>>
```

- Si queremos una **lista con todas las llaves del diccionario**, la podemos obtener así:

```
1 lista_keys = list(diccionario.keys())
```

Por ejemplo:

```
1 lista_keys = list(mi_diccionario.keys())
2 print(lista_keys)
```

```
>>>
[1, 3]
>>>
```

- Si queremos una **lista con todos los valores del diccionario**, la podemos obtener así:

```
1 lista_values = list(diccionario.values())
```

Por ejemplo:

```
1 lista_values = list(mi_diccionario.values())
2 print(lista_values)
```

```
>>>
['palabra1', 'palabra3']
>>>
```

- Por último, si queremos consultar **si una determinada llave k forma parte del diccionario**, lo podemos hacer así:

```
1 var = k in diccionario
```

Si k forma parte del diccionario, `var` tomará el valor **True**, en cambio, si no está en el diccionario, tomará el valor **False**. Por ejemplo:

```
1 var1 = 3 in mi_diccionario
2 print(var1)
3 var2 = 2 in mi_diccionario
4 print(var2)
```

```
>>>
True
False
>>>
```

Ejemplo 8.1.1. Dada una lista de alumnos llamada *lista_alumnos*, donde cada elemento corresponde a un *string* con el número de alumno y nombre, separados por un guión (por ejemplo, '123-Juan'). Crea una función que reciba dicha lista y retorne un diccionario cuyas llaves correspondan al número de alumno y los *values* al respectivo

nombre del alumno. Luego, imprime cada elemento del diccionario de la siguiente forma: "key : value".

```
1 # Definimos la función pedida
2
3 def listaToDict(lista):
4     # Creamos un diccionario vacío
5     diccionario = {}
6     for alumno in lista:
7         # Separamos cada elemento por el guión
8         numero_nombre = alumno.split('-') # Esto es una lista, por ej [123, 'Juan']
9         # Asignamos las variables correspondientes a número de alumno y nombre
10        numero = int(numero_nombre[0])
11        nombre = numero_nombre[1]
12        # Agregamos el elemento al diccionario
13        diccionario[numero] = nombre
14    return diccionario
15
16 # Luego utilizamos esta función sobre la lista dada e imprimimos los values
17
18 diccionario_alumnos = listaToDict(lista_alumnos)
19 for i in list(diccionario_alumnos.keys()):
20     print('{i}: {v}'.format(i, diccionario_alumnos[i]))
```

8.2. Tuplas

Qué es una tupla

Las tuplas son otra estructura de datos y se caracterizan por ser **Inmutables**. Pero, ¿qué significa que sea inmutable?. La idea es que una vez definida la tupla, yo no puedo modificar sus elementos, a diferencia como sucedía con la lista, en donde al definir una lista de la forma `l = [1, 2]` yo podía cambiar un elemento escribiendo `l[1] = 3`. Esto es algo que no podremos hacer con las tuplas.

En Python, podemos definir tuplas escribiendo elementos entre paréntesis, separados por comas. Al igual que con las listas, pueden ser de distinto tipo.

```
1 tupla = (1, '2', True)
```

Uso de tuplas

- Podemos **acceder** a un valor haciendo *slicing*, al igual que como lo hacíamos con los *string* o las listas. Por ejemplo:

```
1 tupla = (1, 2, 'tres', 'cuatro')
2 var1 = tupla[2]
3 var2 = tupla[1:3]
4 print(var1)
5 print(var2)

>>>
tres
(2, 'tres')
```


- Como las tuplas no son mutables, una vez creadas, no podemos cambiar sus valores, ni agregar nuevos, ni eliminar los existentes. Es por esto que no existen métodos de inserción ni de eliminación de valores para las tuplas. Además, si intentamos cambiar un valor, obtendremos un `TypeError`.

```
1 tupla = (1, 2, 'tres', 'cuatro')
2 tupla[2] = 3

>>>
TypeError: 'tuple' object does not support item assignment
>>>
```

- Podemos **recorrer** tuplas.

```
1 tupla = (1, 2, 'tres', 'cuatro')
2 for elemento in tupla:
3     print(elemento)

>>>
1
2
tres
cuatro
>>>
```

- Podemos obtener el **largo** de una tupla.

```
1 tupla = (1, 2, 'tres', 'cuatro')
2 largo = len(tupla)
3 print(largo)

>>>
4
>>>
```

- Podemos determinar **si un elemento** está en una tupla.

```
1 tupla = (1, 2, 'tres', 'cuatro')
2 var1 = 'tres' in tupla
3 var2 = 5 in tupla
4 print(var1)
5 print(var2)

>>>
True
False
>>>
```

- Podemos **unir** dos tuplas con el operador `+`.

```
1 tupla1 = (1, 2, 'tres', 'cuatro')
2 tupla2 = (5, 6, 'siete')
3 tupla = tupla1 + tupla2
4 print(tupla)
```

```
>>>
(1, 2, 'tres', 'cuatro', 5, 6, 'siete')
>>>
```

- Podemos crear una tupla con **un ítem**, pero debemos poner la coma al final del primer elemento.

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

```
>>>
(1,)
<class 'tuple'>
>>>
```

Si no ponemos la coma, entenderá que es un solo elemento y que por lo tanto es un **int** y no una tupla.

```
1 tupla = (1)
2 print(tupla)
3 print(type(tupla))
```

```
>>>
1
<class 'int'>
>>>
```

- Podemos **transformar a tupla** una lista.

```
1 lista = [1, 2, 3, 4]
2 tupla_lista = tuple(lista)
3 print(tupla_lista)
4 print(type(tupla_lista))
```

```
>>>
(1, 2, 3, 4)
<class 'tuple'>
>>>
```

- Podemos **transformar una tupla** a lista.

```
1 tupla = (1, 2, 3, 4)
2 lista_tupla = list(tupla)
3 print(lista_tupla)
4 print(type(lista_tupla))
```

```
>>>
[1, 2, 3, 4]
<class 'list'>
>>>
```

Capítulo 9

Clases

La programación orientada a objetos (OOP por sus siglas en inglés) es el paradigma de programación que se basa en la interacción de **objetos** que nosotros definimos. La idea en general es que ahora tendremos la posibilidad de definir nuestros propios tipos de dato y podremos instanciar elementos de esos tipos que podrán interactuar entre ellos. Para esto vamos a tener que comprender el concepto de **clases**. En este capítulo, aprenderemos cómo crear y utilizar clases en Python para crear nuestros propios tipos de dato y poder usar objetos que son de aquellos tipos. Así, incluso podremos modelar comportamientos de la realidad en nuestros programas.

9.1. Creando una clase

Cuando trabajamos con clases, generalmente es porque queremos modelar el comportamiento de un objeto y su interacción con otros. Por ejemplo, podríamos pensar en la clase **Persona** que posee ciertas características y posibles acciones, que interactúa con otra clase **Auto** que hace otras cosas y que tiene otras características. Utilizaremos este mismo ejemplo para ir aclarando los próximos conceptos y los pasos para la creación de una clase.

Declarar una clase

El primer paso para crear una clase, es declararla y asignarle un nombre. Para esto, recordemos que cuando queríamos definir una función, teníamos la palabra reservada **def** que nos indicaba que lo definido a continuación era una función. Para las clases, utilizamos la palabra reservada **class**, seguida del nombre que le asignaremos a la clase:

```
1 class <Nombre de la clase>:  
2     <Aquí va el contenido de la clase, indentado>
```

La declaración de nuestras clases de ejemplo se verían así:

```
1 class Persona:  
2     # Aquí van las cosas que caracterizan a una persona  
3  
4 class Auto:  
5     # Aquí van las cosas que caracterizan a un auto
```

Observación 9.1.1. Podemos entender las líneas de arriba como que estamos definiendo dos nuevos tipo de dato: Persona y Auto.

Atributos de la clase y constructor

El siguiente paso es definir los atributos de la clase en el constructor. Para esto, primero es importante entender el concepto de **atributo**. Cada clase debe tener ciertas características que definen su comportamiento y que la distinguen de otras clases, estas características son las que nosotros llamaremos atributos. Por ejemplo, consideraremos que las personas tienen rut, nombre, apellido, edad y autos y que los autos tienen marca, modelo y color. Hay que tener en cuenta que, dependiendo de la situación que queramos modelar es que podemos complejizar la clase y agregarle los atributos que consideremos conveniente.

Ahora, hablemos del constructor:

- Es un método (recordemos que los métodos son simplemente funciones) y por lo tanto, utiliza la palabra reservada **def**, a continuación va su nombre: `__init__` (no te asustes porque su nombre comienza y termina con doble guión, es solo su nombre) y finalmente, entre paréntesis y separados por coma van los parámetros que recibe.
- Los parámetros que recibe son las variables con las que **inicializaremos cada nuevo objeto** (esto quedará un poco más claro cuando expliquemos la instanciación de objetos).
- Los atributos de la clase se definen de la siguiente manera: `self.nombre_atributo = valor`.
- Los parámetros se pueden asignar a los atributos y el panorama general se ve más o menos así:

```
1 class <Nombre de la Clase>:  
2  
3     def __init__(self, parametro_1,..., parametro_n):  
4         self.atributo_1 = parametro_1  
5         self.atributo_2 = parametro_2  
6         #...  
7         self.atributo_n = parametro_n
```

Continuando con nuestro ejemplo, completemos los constructores de personas y autos:

```
1 class Persona:  
2     def __init__(self, rut, nombre, apellido, edad_ingresada):  
3         self.rut = rut  
4         self.nombre = nombre  
5         self.apellido = apellido  
6         self.edad = edad_ingresada  
7         self.autos = []  
8  
9 class Auto:  
10    def __init__(self, marca, modelo, color):  
11        self.marca = marca  
12        self.modelo = modelo  
13        self.color = color  
14        self.dueno = None
```

Es muy importante notar lo siguiente:

- Un atributo **toma el valor** de un parámetro ingresado, por lo que no necesariamente deben tener el mismo nombre. Por ejemplo, en la clase `Persona`, el atributo es `edad`, pero el parámetro es `edad_ingresada`.
- Los parámetros no necesariamente tienen que ser asignados a atributos. Se puede trabajar con ellos dentro del constructor (como en una función normal).

- Un atributo no necesariamente tiene que tomar el valor entregado por un parámetro. Por ejemplo, en la clase `Persona`, tenemos el atributo `autos` que no toma el valor de ninguno de los parámetros, pero nosotros le asignamos como valor inicial una lista vacía. Esto quiere decir que, inicialmente, todas las personas van a iniciar con ningún auto (o dicho de otra manera, nadie tiene ningún auto al comienzo), a diferencia de los otros atributos que comenzarán inicializados con los valores entregados por los parámetros. Esto también ocurre con el atributo `dueno` de la clase `Auto`, que no toma el valor de ningún parámetro y comienza inicializado como `None`, lo que quiere decir que, inicialmente, los autos instanciados no tendrán un dueño asociado.

9.2. Agregando métodos

Al crear una clase, deberíamos tener por lo menos el método del constructor, que será el encargado de inicializar los atributos una vez que se instancia el objeto. Pero la parte interesante, es que podemos agregarle más métodos a nuestra clase, dependiendo del comportamiento que queramos modelar. Para esto, agregamos al nivel de indentación del constructor los nuevos métodos:

```

1 class <Nombre de la Clase>:
2
3     def __init__(self, parametro_1,..., parametro_n):
4         self.atributo_1 = parametro_1
5         self.atributo_2 = parametro_2
6         #...
7         self.atributo_n = parametro_n
8
9     def metodo_1(self, parametro_1,...,parametro_n):
10        # Aquí va el contenido del método 1
11
12    def metodo_2(self, parametro_1,...,parametro_n):
13        # Aquí va el contenido del método 2

```

Es muy importante notar que, cada método definido en una clase debe recibir un primer parámetro que puede llevar cualquier nombre, pero es una buena práctica llamarlo `self`. Este parámetro nos sirve para hacer referencia al objeto mismo y así acceder a sus atributos y métodos. En el constructor, cuando queríamos asignar valores al atributo del objeto debíamos utilizar `self` para su definición, si no lo hubiésemos usado, esa variable no habría quedado almacenada en el objeto.

Continuaremos con el ejemplo de las personas y autos agregando algunos métodos:

```

1 # Agregamos a la clase Persona los métodos cumpleaños, saludar y comprar_auto
2 class Persona:
3     def __init__(self, rut, nombre, apellido, edad_ingresada):
4         self.rut = rut
5         self.nombre = nombre
6         self.apellido = apellido
7         self.edad = edad_ingresada
8         self.autos = []
9
10    def cumpleaños(self):
11        # Incrementamos la edad de la persona en 1
12        # Notemos que este método no recibe parámetros, aparte de self
13        self.edad += 1
14
15    def saludar(self):
16        # Imprimimos un saludo utilizando los datos de la persona

```

```

17     print('Hola! mi nombre es {} {}'.format(self.nombre, self.apellido))
18
19     def comprar_auto(self, auto):
20         # Agregamos un objeto de tipo/clase Auto a la lista de autos de la persona
21         self.autos.append(auto)
22         # Modificamos al dueño del auto
23         # Asumimos que el parámetro auto será de clase Auto, por lo que tendrá el atributo dueño
24         auto.dueno = self
25
26 # Agregamos a la clase Auto el método andar
27 class Auto:
28     def __init__(self, marca, modelo, color):
29         self.marca = marca
30         self.modelo = modelo
31         self.color = color
32         self.dueno = None
33
34     def andar(self):
35         # Imprimimos algo que nos indique que el auto está en movimiento
36         print('Me estoy moviendo')

```

9.3. Utilizando una clase

Para poder utilizar de manera correcta las clases, necesitamos primero entender la diferencia entre su definición y su instanciación. Hasta ahora, lo único que hemos hecho es definir clases (Personas y Autos), esto es más o menos equivalente a definir el concepto de Persona y el concepto de Auto y establecer sus características y posibles acciones. Pero ahora necesitamos los objetos particulares que nos permitirán modelar situaciones, es decir, necesitamos una persona concreta y un auto en concreto. Dicho de otra forma, por ahora solo sabemos las características de una persona (definidas por su clase), pero necesitamos a un Juan y a una María.

Instanciando un objeto

Para instanciar un objeto de cierta clase, hacemos lo siguiente:

```
1 mi_objeto = Clase(parametro_1, ..., parametro_n)
```

En donde mi_objeto es una instancia de la clase Clase. Continuando con nuestro ejemplo de personas y autos:

```

1 # Definición de clases
2
3 class Persona:
4     def __init__(self, rut, nombre, apellido, edad_ingresada):
5         self.rut = rut
6         self.nombre = nombre
7         self.apellido = apellido
8         self.edad = edad_ingresada
9         self.autos = []
10
11     def cumpleaños(self):
12         self.edad += 1
13
14     def saludar(self):
15         print('Hola! mi nombre es {} {}'.format(self.nombre, self.apellido))

```

```

16
17 def comprar_auto(self, auto):
18     self.autos.append(auto)
19     auto.dueno = self
20
21 class Auto:
22     def __init__(self, marca, modelo, color):
23         self.marca = marca
24         self.modelo = modelo
25         self.color = color
26         self.dueno = None
27
28     def andar(self):
29         print('Me estoy moviendo')
30
31 # Inicio del programa, lugar donde instanciamos
32
33 p1 = Persona('1.234.567-8', 'Juan', 'Pérez', 30)
34 p2 = Persona('2.345.678-9', 'María', 'Saldías', 23)
35 p3 = Persona('3.456.789-k', 'Marcelo', 'Riveros', 15)
36 a1 = Auto('Hyundai', 'i30', 'Blanco')
37 a2 = Auto('Susuki', 'Vitara', 'Rojo')
38
39 print(p1.nombre)
40 print(p2.nombre)
41 print(p3.nombre)
42 print(p1.edad)
43 print(p2.edad)
44 print(a1.marca)
45 print(a2.color)
46 print(type(p1))
47 print(type(p2))
48 print(type(a2))

```

Al ejecutarlo, obtenemos lo siguiente:

```

>>>
Juan
María
Marcelo
30
23
Hyundai
Rojo
<class '__main__.Persona'>
<class '__main__.Persona'>
<class '__main__.Auto'>
>>>

```

Del código anterior, notemos que:

- p1 y p2 son personas diferentes, uno es Juan Pérez y la otra es María Saldías y ambos se instancian a partir de la misma clase `Persona`.
- Podemos acceder a los atributos de una instancia de la forma `instancia.atributo`.

- Al instanciar, solo entregamos los parámetros que no son **self**. Por ejemplo, en la definición de la clase `Persona` tenemos que recibe los parámetros: `self`, `rut`, `nombre`, `apellido` y `edad`, pero al instanciar a una persona en particular sólo entregamos `rut`, `nombre`, `apellido` y `edad`.
- Si aplicamos la función `type()` a los objetos, vemos que nos indica la clase a la que pertenecen.

Usando los métodos

Los métodos se utilizan para una instancia en particular y podemos llamarlos así: `objeto.metodo(parametros)`

```

1 # Definición de clases
2
3 class Persona:
4     def __init__(self, rut, nombre, apellido, edad_ingresada):
5         self.rut = rut
6         self.nombre = nombre
7         self.apellido = apellido
8         self.edad = edad_ingresada
9         self autos = []
10
11     def cumpleaños(self):
12         self.edad += 1
13
14     def saludar(self):
15         print('Hola! mi nombre es {} {}'.format(self.nombre, self.apellido))
16
17     def comprar_auto(self, auto):
18         self.autos.append(auto)
19         auto.dueno = self
20
21 class Auto:
22     def __init__(self, marca, modelo, color):
23         self.marca = marca
24         self.modelo = modelo
25         self.color = color
26         self.dueno = None
27
28     def andar(self):
29         print('Me estoy moviendo')
30
31 # Instanciamos
32
33 p1 = Persona('1.234.567-8', 'Juan', 'Pérez', 30)
34 p2 = Persona('2.345.678-9', 'María', 'Saldías', 23)
35 p3 = Persona('3.456.789-k', 'Marcelo', 'Riveros', 15)
36 a1 = Auto('Hyundai', 'i30', 'Blanco')
37 a2 = Auto('Susuki', 'Vitara', 'Rojo')
38
39 # Utilizamos los métodos
40
41 p1.saludar()
42 p3.saludar()
43 print('Edad antes del cumpleaños: ', p2.edad)
44 p2.cumpleaños()
45 print('Edad post-cumpleaños: ', p2.edad)

```



```

46 print('Autos de ', p1.nombre, ' antes de comprar: ', p1.autos)
47 p1.comprar_auto(a1)
48 print('Autos de ', p1.nombre, ' post compra: ', p1.autos)
49 print('El dueño del auto a1 es: ', a1.dueno)
50 print('El dueño del auto a1 es: ', a1.dueno.nombre)
51 a1.andar()

```

Ejecutando, obtenemos lo siguiente:

```

>>>
Hola! mi nombre es Juan Pérez
Hola! mi nombre es Marcelo Riveros
Edad antes del cumpleaños: 23
Edad post-cumpleaños: 24
Autos de Juan antes de comprar: []
Autos de Juan post compra: [<__main__.Auto object at 0x02D6BE90>]
El dueño del auto a1 es: <__main__.Persona object at 0x02BC15B0>
El dueño del auto a1 es: Juan
Me estoy moviendo
>>>

```

En este caso, hay que destacar que:

- Al igual que para los atributos, solo se pasan los parámetros que no son **self**. Por ejemplo, el método `saludar` sólo recibía **self**, por lo que no le pasamos ningún parámetro. En el caso del método `comprar_auto`, recibía **self** y `auto`, por lo que sólo le entregamos `auto`.
- Existen buenas prácticas relacionadas al acceso de métodos y atributos de las diferentes clases. Esto tiene relación con que los atributos y métodos pueden ser públicos o privados, pero no deberías preocuparte por ahora, pues es materia de cursos más avanzados, aunque es bueno que lo tengas en cuenta.
- Debes haberte fijado que se ve un poco extraño y poco intuitivo la forma en que se imprimió la lista de autos de Juan que contenía un objeto de clase `Auto`. Esto es porque no hemos definido una forma en que queremos que se represente o imprima dicho objeto, entonces se muestra el espacio de memoria en que está ubicada la instancia. Esto lo comentaremos en la siguiente sección.

9.4. Imprimiendo clases

Como mencionamos anteriormente, si no definimos una manera de representar y/o imprimir un objeto, este se verá similar a lo siguiente: `<__main__.Auto object at 0x02D6BE90>`. Para poder imprimir un objeto de una manera más intuitiva y legible, podemos utilizar los métodos `__str__` o `__repr__`. Para explicar estos métodos tomemos en consideración el siguiente ejemplo simplificado de `Personas` y `Autos`:

```

1 class Persona:
2     def __init__(self, nombre, edad):
3         self.nombre = nombre
4         self.edad = edad
5         self.autos = []
6
7     def comprar_auto(self, auto):
8         self.autos.append(auto)
9         auto.dueno = self
10
11 class Auto:

```

```

12 def __init__(self, modelo, color):
13     self.modelo = modelo
14     self.color = color
15     self.dueno = None

```

Método `__str__`

Agregamos a ambas clases el método `__str__` que recibe como parámetro `self` y retorna un *string* con lo que queremos que represente al objeto, en el caso de `Personas`, retornaremos su nombre y en el caso de `Autos`, el modelo, color y dueño, separados por comas.

```

1 class Persona:
2     def __init__(self, nombre, edad):
3         self.nombre = nombre
4         self.edad = edad
5         self.autos = []
6
7     def comprar_auto(self, auto):
8         self.autos.append(auto)
9         auto.dueno = self
10
11     def __str__(self):
12         # Retornamos el nombre de la persona
13         return self.nombre
14
15 class Auto:
16     def __init__(self, modelo, color):
17         self.modelo = modelo
18         self.color = color
19         self.dueno = None
20
21     def __str__(self):
22         # Retornamos modelo, color, dueno
23         return '{}, {}, {}'.format(self.modelo, self.color, self.dueno)
24
25 # Instanciamos
26 p1 = Persona('Juan', 30)
27 p2 = Persona('María', 25)
28 a1 = Auto('i30', 'blanco')
29
30 # Imprimimos
31 print(a1)
32 print(p1)
33 print(p2)
34 p1.comprar_auto(a1)
35 print(p1.autos)
36 print(a1.dueno)

```

Obtenemos lo siguiente al ejecutar el código anterior:

```

>>>
i30, blanco, None
Juan
María

```

```
[<__main__.Auto object at 0x034562F0>]
Juan
>>>
```

Si bien observamos que cuando imprimimos una instancia obtenemos lo que retorna el método `__str__`, vemos que cuando el objeto está dentro de una lista, esto no funciona. En esos casos podemos utilizar `__repr__`.

Método `__repr__`

Agregamos de manera similar a `__str__`, el método `__repr__` en ambas clases.

```
1 class Persona:
2     def __init__(self, nombre, edad):
3         self.nombre = nombre
4         self.edad = edad
5         self.autos = []
6
7     def comprar_auto(self, auto):
8         self.autos.append(auto)
9         auto.dueno = self
10
11     def __repr__(self):
12         return self.nombre
13
14 class Auto:
15     def __init__(self, modelo, color):
16         self.modelo = modelo
17         self.color = color
18         self.dueno = None
19
20     def __repr__(self):
21         return '{}, {}, {}'.format(self.modelo, self.color, self.dueno)
22
23 #Instanciamos
24 p1 = Persona('Juan', 30)
25 p2 = Persona('María', 25)
26 a1 = Auto('i30', 'blanco')
27
28 print(a1)
29 print(p1)
30 p1.comprar_auto(a1)
31 print(p1.autos)
32 print(a1.dueno)
33
34 >>>
35 i30, blanco, None
36 Juan
37 [i30, blanco, Juan]
38 Juan
39 >>>
```

9.5. Atributos de instancia y de clase

Hasta ahora, solo hemos trabajado con atributos de instancia, pues estos se encontraban dentro del constructor de la instancia. Pero también existen los atributos de clase, que son compartidos por todas las instancias de esa clase. Por ejemplo, tomemos la siguiente clase `Persona`:

```
1 class Persona:
2     def __init__(self, nombre, edad):
3         self.nombre = nombre
4         self.edad = edad
5         self.tributo = 'valor'
6         otro_tributo = 'otro_valor'
7
8 p1 = Persona('Juan', 30)
9 p2 = Persona('María', 25)
10 p3 = Persona('Marcelo', 15)
11
12 # Imprimimos
13 print('---Antes del cambio---')
14 print(p1.tributo)
15 print(p2.tributo)
16 print(p3.tributo)
17
18 p1.tributo = 'nuevo_valor'
19 print('---Post cambio---')
20 print(p1.tributo)
21 print(p2.tributo)
22 print(p3.tributo)
```

Al ejecutar, tenemos lo siguiente:

```
>>>
---Antes del cambio---
valor
valor
valor
---Post cambio---
nuevo_valor
valor
valor
>>>
```

Como hemos modificado el valor de un atributo en una instancia, solo se modifica para esa instancia y no para las demás. Ahora, podríamos pensar que un atributo que no está definido con `self` (como `otro_tributo`) podría ser compartido por todos los objetos de la clase. Pero si intentamos acceder a él, vemos que el programa se cae.

```
1 class Persona:
2     def __init__(self, nombre, edad):
3         self.nombre = nombre
4         self.edad = edad
5         self.tributo = 'valor'
6         otro_tributo = 'otro_valor'
7
8 p1 = Persona('Juan', 30)
9 p2 = Persona('María', 25)
```

```

10 p3 = Persona('Marcelo', 15)
11
12 # Imprimimos
13 print('---Antes del cambio---')
14 print(p1.otro_atributo)
15 print(p2.otro_atributo)
16 print(p3.otro_atributo)
17 p1.otro_atributo = 'nuevo_otro_valor'
18 print('---Post cambio---')
19 print(p1.otro_atributo)
20 print(p2.otro_atributo)
21 print(p3.otro_atributo)

>>>
AttributeError: 'Persona' object has no attribute 'otro_atributo'
>>>

```

Entonces ¿cómo podemos definir un atributo que sea **de la clase**? Lo podemos hacer definiendo el atributo luego de la declaración de la clase y antes del constructor:

```

1 class Persona:
2
3     atributo_clase = 'a'
4
5     def __init__(self, nombre, edad):
6         self.nombre = nombre
7         self.edad = edad
8         self.atributo = 'valor'
9
10 p1 = Persona('Juan', 30)
11 p2 = Persona('María', 25)
12 p3 = Persona('Marcelo', 15)
13
14 # Imprimimos
15 print('---Antes del cambio---')
16 print(p1.atributo_clase)
17 print(p2.atributo_clase)
18 print(p3.atributo_clase)
19 Persona.atributo_clase = 'b'
20 print('---Post cambio en clase Persona---')
21 print(p1.atributo_clase)
22 print(p2.atributo_clase)
23 print(p3.atributo_clase)
24 p1.atributo_clase = 'c'
25 print('---Post cambio en instancia p1---')
26 print(p1.atributo_clase)
27 print(p2.atributo_clase)
28 print(p3.atributo_clase)

>>>
---Antes del cambio---
a
a
a
---Post cambio en clase Persona---

```

```

b
b
b
---Post cambio en instancia p1---
c
b
b
>>>

```

Vemos al comienzo que si accedemos a este atributo, de la forma `instancia.atributo`, todas las instancias comparten este valor. Ahora, si modificamos su valor de la forma `clase.atributo = nuevo_valor` este se cambia para todas las instancias. Pero si modificamos su valor de la forma `instancia.atributo = nuevo_valor` se cambia solo para esa instancia y el antiguo valor permanece en los otros objetos.

Si bien es muy importante entender la diferencia entre atributos de clase y de instancia para evitar problemas de modelación, esto también nos puede ser de mucha utilidad. Supongamos que queremos que las personas tengan un atributo `identificador` que puede ser implementado como un contador y asignar así al `id` el número que va incrementando. Una manera fácil de hacerlo, sería con una variable global que se incrementa cada vez que se instancia la clase, pero una mejor forma de modelarlo sería usando variables de clase de la siguiente manera:

```

1 class Persona:
2
3     identificador = 0
4
5     def __init__(self, nombre, edad):
6         self.identificador = Persona.identificador + 1
7         Persona.identificador += 1
8         self.nombre = nombre
9         self.edad = edad
10        self.atributo = 'valor'
11
12 p1 = Persona('Juan', 30)
13 p2 = Persona('María', 25)
14 p3 = Persona('Marcelo', 15)
15
16 print(p1.identificador)
17 print(p2.identificador)
18 print(p3.identificador)
19 print(Persona.identificador)

```

Al ejecutar, podemos comprobar que cada persona tiene su propio `identificador`, comenzando desde 1. El atributo de clase queda seteado en el valor 3 luego de haber instanciado 3 objetos y si volviéramos a instanciar la clase, seguiría incrementando. Notemos que esto funciona debido a que al hacer `self.identificador`, el atributo pasa a ser de la instancia y deja de ser compartido por la clase, por lo que si el atributo de la clase cambia (al instanciarse otro objeto), en esta instancia no se volverá a cambiar.

```

>>>
1
2
3
3
>>>

```

Veamos ahora un ejemplo de un error típico al intentar implementar lo anterior:

```

1 class Persona:
2
3     identificador = 0
4
5     def __init__(self, nombre, edad):
6         self.id = Persona.identificador
7         Persona.identificador += 1
8         self.nombre = nombre
9         self.edad = edad
10        self. atributo = 'valor'
11
12 p1 = Persona('Juan', 30)
13 p2 = Persona('María', 25)
14 p3 = Persona('Marcelo', 15)
15
16 print(p1.identificador)
17 print(p2.identificador)
18 print(p3.identificador)
19 print(Persona.identificador)

```

```

>>>
3
3
3
3
>>>

```

En este caso, le asignamos al atributo de instancia `id` una **referencia** al valor del atributo de clase `identificador` y luego incrementamos el valor del atributo de clase. El error aquí es que, como mencionamos, `id` hace referencia al valor de `Persona.identificador`, por lo que si `Persona.identificador` cambia, también lo hará el `id`.

9.6. Ejercicios propuestos

Ejercicio 9.6.1. Modela una cola de supermercado que cuenta con una sola caja teniendo en consideración lo siguiente:

- Los clientes llegan en un tiempo aleatorio entre 7 y 12 minutos.
- La caja tarda en atender al cliente entre 10 y 15 minutos.
- La caja está abierta durante 480 minutos y debes simularla por 5 días.

Queremos saber cuántos clientes en promedio no son atendidos en un día, cuánto se demora un cliente en promedio y el promedio de gente atendida por día.

Ejercicio 9.6.2. Hoy en día los fraudes bancarios han crecido exponencialmente. Muchas veces los clientes de un banco no se daban cuenta porque no tenían acceso a revisar su estado de cuenta a través de internet. Para dar solución a este problema, crea un programa en Python que permita llevar la administración de las cuentas corrientes de los clientes del *Banco de Catán*.

El programa debe permitir almacenar clientes y sus transacciones bancarias: nombre, rut, género, balance y lista de transacciones. El atributo lista de transacciones a su vez, está compuesto por tuplas de 4 elementos cada una:

tipo transacción, fecha, descripción y monto, de esta forma un cliente puede tener muchas transacciones registradas. El formato de la fecha es dd-mm-yyyy, donde dd es día, mm es mes y yyyy es año (por ejemplo, 21-03-2014). El atributo balance de un cliente, determina el monto inicial que un cliente tiene en su cuenta bancaria. Los tipos de transacción válidos son: Depósito, Giro y Compra.

El programa debe permitir crear nuevos clientes y administrar su información. Por lo tanto, se debe contar con el siguiente menú principal:

```
>>>
Bienvenido al Sistema Banco de Catán
Seleccione una acción:
[1] Crear cliente
[2] Mostrar clientes
[3] Buscar clientes
[4] Agregar transacción a un cliente
[5] Salir
>>>
```

- La **opción [1]** del menú principal debe permitir al usuario crear un nuevo cliente.
- La **opción [2]** del menú principal debe permitir desplegar todos los clientes que están registrados en el banco. Por ejemplo, si se han registrado 2 clientes, se debe mostrar la información completa de cada uno:

```
>>>
Clientes:
[1] Julio Meneses, 1.456.273-5, Masculino,
    [('Depósito', '15-01-2010', 'para gastos comunes', 1000),
     ('Compra', '11-03-2012', 'zapatos Merrel', 50000)]
[2] Fabiola Retamal, 19.264.198-4, Femenino,
    [('Giro', '05-19-2013', 'para préstamo', 17000),
     ('Giro', '11-04-2014', 'compra cartera', 350000)]
>>>
```

- La **opción [3]** del menú principal debe mostrar el siguiente sub-menú:

```
>>>
Ingrese el tipo de búsqueda:
[1] Por nombre de cliente
[2] Por nombre de cliente y tipo de transacción
[3] Por fecha
[4] Salir
>>>
```

La opción [1] debe preguntar por el nombre del cliente y desplegar el registro completo del cliente.

La opción [2] debe preguntar por el nombre del cliente y el tipo de transacción (Depósito, Giro o Compra). De esta forma, se desplegarán todas las transacciones de cierto tipo de un cliente determinado.

La opción [3] debe mostrar la información de todos los clientes que tengan transacciones en la fecha ingresada. Es decir, el nombre del cliente, el rut y la tupla de la transacción con la fecha buscada.

La opción [4] debe volver al menú principal.

- La **opción [4]** del menú principal debe permitir agregar una nueva transacción a un cliente. Para eso, se deben desplegar todos los clientes, seleccionar uno de ellos y finalmente agregar la transacción. Por ejemplo, si se selecciona esta opción, se despliegan los cliente hasta ahora registrados:

```
>>>
Seleccione cliente:
[1] Julio Meneses, 1.456.273-5, Masculino,
    [('Depósito', '15-01-2010', 'para gastos comunes', 1000),
     ('Compra', '11-03-2012', 'zapatos Merrel', 50000)]
[2] Fabiola Retamal, 19.264.198-4, Femenino,
    [('Giro', '05-19-2013', 'para préstamo', 17000),
     ('Giro', '11-04-2014', 'compra cartera', 350000)]
2
Ingrese tipo transacción: Compra
Ingrese fecha transacción: 01-04-2014
Ingrese descripción: Polera Guchi
Ingrese monto de la transacción: 110000
Se ha agregado una transacción al cliente Fabiola Retamal:
    [('Giro', '05-19-2013', 'para préstamo', 17000),
     ('Giro', '11-04-2014', 'compra cartera', 350000),
     ('Compra', '01-04-2014', 'Polera Guchi', 11000)]
>>>
```

- La **opción [5]** del menú principal permite salir completamente del programa de Sistema Banco de Catán.

Se te pide que crees la clase Cliente que debe contar con los siguientes atributos:

- Un *string* para el nombre del cliente
- Un *string* para el rut del cliente
- Un *string* para la fecha de la transacción
- Un entero para el balance del cliente
- Una lista para las transacciones del cliente (recuerda que cada transacción será una tupla con (tipo transacción, fecha transacción, descripción, monto))

Además, la clase cliente debe contar al menos con los siguientes métodos:

- `agregar_transaccion(self, nueva_transaccion)`: este método permite agregar una nueva transacción al listado de transacciones que ya tiene un cliente.
- `verifica_balance(self)`: este método retorna **True** si el cliente aún tiene balance positivo, y **False** en caso de tener balance negativo.
- `actualiza_balance(self, monto)`: este método actualiza el balance de un cliente utilizando el parámetro `monto`, que recibe el método.

Tu programa principal debe contar con al menos las siguientes funciones:

- `mostrar_menu_principal()`: esta función solo despliega el menú principal.
- `mostrar_clientes(lista_clientes)`: esta función debe desplegar todos los clientes del banco.

- `buscar_cliente(lista_clientes, cliente_a_buscar)`: esta función recibe la `lista_clientes` y el `cliente_a_buscar`. La función debe mostrar toda la información que contenga el `cliente_a_buscar`.
- `buscar_transaccion_en_cliente(lista_clientes, transaccion_a_buscar, cliente_a_buscar)`: esta función debe mostrar el nombre y rut del `cliente_a_buscar` y además todas las transacciones que sean del tipo `transaccion_a_buscar` de ese cliente.
- `buscar_fecha_transaccion(lista_clientes, fecha_a_buscar)`: esta función debe mostrar todos los clientes que tengan transacciones en la `fecha_a_buscar`.

En el caso de que se esté buscando y no se encuentren coincidencias, se debe mostrar el mensaje “no existen coincidencias”.

Capítulo 10

Archivos

Hasta ahora, todo lo que hemos programado no persiste en el tiempo, es decir, una vez cerrado el programa, toda interacción con el usuario no tendrá efecto en las posteriores ejecuciones. Pero, ¿qué pasa si queremos guardar permanentemente los datos entregados por nuestro programa?. Para esto se hace necesario hacer uso del manejo de archivos. En este capítulo nos enfocaremos en leer y escribir archivos de texto plano (archivos con extensión `.txt`).

10.1. Ruta de un archivo

La ruta (o *path*) de un archivo es lo que nos indica su ubicación en el computador y esta puede ser de tipo relativa o absoluta. Observemos la siguiente imagen para entender mejor la diferencia entre ambas:

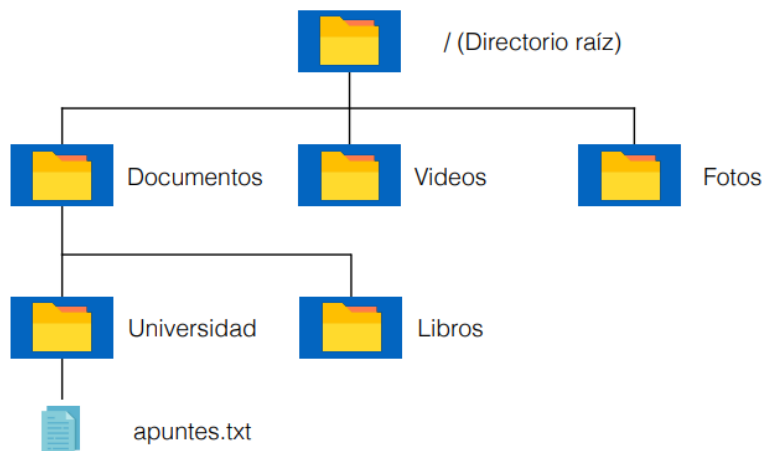


Figura 10.1: ejemplo de rutas.

Una ruta **absoluta** indica una *path* independiente de donde estemos “parados” ejecutando. Por ejemplo:

`/Documentos/Universidad/apuntes.txt`.

Una ruta **relativa** depende de la ubicación del `.py` que ejecutamos. Por ejemplo, si estuviéramos “parados” en `Documentos` y quisiéramos llegar a `apuntes.txt`, la ruta relativa sería **`./Universidad/apuntes.txt`**, pero si estuviéramos “parados” en `Libros`, la ruta relativa para llegar a `apuntes.txt` sería **`../Universidad/apuntes.txt`**.

Observación 10.1.1. Cuando utilizamos un punto (`./`), es lo mismo que hablar del archivo donde estamos “parados”. Esto puede ser simplificado y no ser usado, por ejemplo, la ruta relativa `./Universidad/apuntes.txt` es equivalente a escribir `Universidad/apuntes.txt` y la ruta `../Universidad/apuntes.txt` a `../Universidad/apuntes.txt`

Observación 10.1.2. Cuando utilizamos doble punto (`../`), nos estamos refiriendo a la carpeta padre de donde estamos “parados”. Entonces, en el ejemplo en que estamos en `Libros`, con `../` volvemos a `Documentos`, luego nos vamos a `Universidad` y finalmente llegamos a `apuntes.txt`.

Para el manejo de rutas es necesario importar el módulo `os`, pero esto es tema de cursos más avanzados. Por ahora nos enfocaremos en trabajar con archivos que se encuentran en la misma carpeta que nuestro programa (es decir, nuestro archivo con extensión `.py`) y usaremos su ruta relativa.

10.2. Abrir y cerrar un archivo de texto

Antes de leer o escribir sobre un archivo de texto, necesitamos aprender a **abrir** archivos. Esto lo podemos hacer con la función `open()`, que recibe como parámetros la ruta donde se encuentra el archivo y el modo (lectura o escritura). Esta función retorna un objeto de tipo `file` que debemos almacenar en una variable.

```
1 mi_archivo = open('<nombre del archivo>', '<modo>')
```

Recordemos que el archivo se encuentra en la misma carpeta que nuestro programa, por lo que la ruta del archivo necesita solo su nombre. También hay que destacar que el nombre del archivo debe incluir su extensión `.txt` y tanto el nombre como el modo deben ser escritos entre comillas.

Algo que es muy importante y no debemos olvidar, es que luego de usar un archivo (ya sea por lectura o escritura) es necesario **cerrarlo**, porque se pueden generar diversos problemas si no lo hacemos: a veces los archivos solo pueden ser abiertos por un programa a la vez o puede que lo que se haya escrito no quede guardado hasta que el archivo se haya cerrado o incluso porque el límite en la cantidad de archivos que puede manejar un programa puede ser bajo. Para cerrar un archivo basta con utilizar la función `close()`.

```
1 mi_archivo.close()
```

Otra opción existente para abrir y cerrar archivos y que es equivalente a usar `open()`, `close()` y guardar el objeto tipo `file` en la variable `mi_archivo` es la siguiente:

```
1 with open('<nombre del archivo>', '<modo>') as mi_archivo:  
2     # Aquí va, intentado, la lectura y/o escritura del archivo
```

10.3. Leer un archivo de texto

Para leer un archivo de texto, necesitamos abrir un archivo en modo lectura, lo que se denota con `'r'` (`read`). Luego, necesitamos funciones que lean el objeto de tipo `file` generado. Aquí podemos usar la función `readline` o la función `readlines`.

Readline

La función `readline` lee la primera línea del archivo cargado (identifica esta línea según los saltos de línea) y la retorna como `string` (generalmente con el fin de almacenarlo en una variable), además, esta línea se elimina del archivo cargado (no del archivo original). Supongamos que tenemos un archivo llamado `archivo.txt` que contiene el siguiente texto:

```
Soy un archivo
Esta es la segunda línea
Y esta es la tercera
```

Entonces, veamos como se aplica la función:

```
1 # Abrimos el archivo
2 mi_archivo = open('archivo.txt', 'r')
3
4 # Leemos
5 linea_1 = mi_archivo.readline()
6 print(linea_1)
7 linea_2 = mi_archivo.readline()
8 print(linea_2)
9 linea_3 = mi_archivo.readline()
10 print(linea_3)
11
12 # Cerramos el archivo
13 mi_archivo.close()
```

```
>>>
```

```
Soy un archivo
```

```
Esta es la segunda línea
```

```
Y esta es la tercera
```

```
>>>
```

Con el primer llamado de la función se lee “Soy un archivo“ y se borra, por lo tanto ahora la primera línea es “Esta es la segunda línea“ y así sucesivamente podemos leer todo el archivo. También notamos que el programa no pudo leer el tilde presente en “línea“ y esto tiene que ver con la manera en que se codifican/decodifican los archivos en tu computador, por lo que puede que no tengas este problema. Pero en caso de que suceda, se puede solucionar de la siguiente forma: agregando el parámetro `encoding` a la función `open()`, con valor `'utf-8'`.

```
1 # Abrimos el archivo
2 mi_archivo = open('archivo.txt', 'r', encoding='utf-8')
3
4 # Leemos
5 linea_1 = mi_archivo.readline()
6 print(linea_1)
7 linea_2 = mi_archivo.readline()
8 print(linea_2)
9 linea_3 = mi_archivo.readline()
10 print(linea_3)
11
12 # Cerramos el archivo
13 mi_archivo.close()
```

```
>>>
```

```
Soy un archivo
```

```
Esta es la segunda línea
```

```
Y esta es la tercera
>>>
```

Readlines

La función `readlines` lee todas líneas y las retorna como una lista de *strings* (generalmente para almacenarlas en otra variable) y las borra del archivo cargado. Aquí es importante tener cuidado con el tamaño del archivo que estamos leyendo, pues si es muy grande para guardarlo en memoria, el programa nos arrojará un error. Los *string* retornados en la lista incluyen los saltos de línea si es que los tenían.

```
1 # Abrimos el archivo
2 mi_archivo = open('archivo.txt', 'r', encoding='utf-8')
3
4 # Leemos
5 lineas = mi_archivo.readlines()
6 print(lineas)
7
8 # Cerramos el archivo
9 mi_archivo.close()

>>>
['Soy un archivo\n', 'Esta es la segunda línea\n', 'Y esta es la tercera']
>>>
```

Vemos que `lineas` es una lista donde cada elemento corresponde a una línea del archivo (con tipo *string*, incluyendo su salto de línea). Es recomendado usar esta función, ya que deja todo almacenado en una sola lista y es más sencillo trabajar sobre listas que sobre objetos tipo *file*.

Ejemplo 10.3.1. Imprime en consola, como un solo *string* y sin saltos de línea, el contenido del archivo dado `archivo_ejemplo.txt`:

```
Hola,
soy
un
archivo.
```

La solución propuesta es la siguiente:

```
1 # Creamos la variable que irá guardando el contenido del archivo. Comienza como string vacío.
2 s = ''
3
4 # Abrimos el archivo
5 archivo = open('archivo_ejemplo.txt', 'r', encoding='utf-8')
6
7 # Leemos el archivo
8 lineas = archivo.readlines()
9
10 # Recorremos la lista de líneas, quitamos el salto de línea y lo agregamos al string s
11 for linea in lineas:
12     s = s + linea.strip('\n')
13
14 # Imprimimos
15 print(s)
```

```

16
17 # Cerramos el archivo
18 archivo.close()

```

El código anterior es equivalente a:

```

1 # Creamos la variable que irá guardando el contenido del archivo. Comienza como string vacío
2 s = ''
3
4 # Abrimos el archivo
5 with open('archivo_ejemplo.txt', 'r', encoding='utf-8') as archivo:
6     # Leemos el archivo
7     lineas = archivo.readlines()
8     # Recorremos la lista de líneas, quitamos el salto de línea y lo agregamos al string s
9     for linea in lineas:
10        s = s + linea.strip('\n')
11
12 # Imprimimos
13 print(s)

```

Si ejecutamos cualquiera de los dos, verificamos que se cumple lo pedido:

```

>>>
Hola, soy un archivo.
>>>

```

10.4. Escribir un archivo de texto

Para escribir un archivo de texto, necesitamos abrir un archivo en modo escritura, lo que se puede denotar con **'w'** (*write*) o con **'a'** (*append*).

Modo write

En este modo, si el archivo no existe, Python lo creará. Si el archivo ya existe, se borrará su contenido y se comenzará a escribir de nuevo, es decir, **se sobrescribirá**. Utilizamos **'w'** en la función **open()** y además, necesitamos la función **write()**, que recibe como parámetro un *string* con el contenido que queremos escribir en el archivo.

```

1 mi_archivo = open('<Nombre del archivo>', 'w')
2 mi_archivo.write(<String con contenido a escribir>)
3 mi_archivo.close()

```

O su equivalente:

```

1 with open('<Nombre del archivo>', 'w') as mi_archivo:
2     mi_archivo.write(<String con contenido a escribir>)

```

Veamos su aplicación en el siguiente ejemplo. No olvidemos incorporar el parámetro **encoding** para evitar problemas con los caracteres como las tildes u otros:

```

1 mi_archivo = open('archivo.txt', 'w', encoding='utf-8')
2 mi_archivo.write('Escribe esto en el documento\nEsta va a ser otra línea\nY esta otra')
3 mi_archivo.close()

```

Como el archivo `archivo.txt` no existía, estamos creándolo en la misma carpeta de nuestro programa (recordemos que estamos trabajando con rutas relativas) y además le estamos agregando contenido. Notemos que el *string* posee el caracter `'\n'`, que significa salto de línea. Ahora, si abrimos el archivo, veremos lo siguiente:

```
Escribe esto en el documento
Esta va a ser otra línea
Y esta otra
```

Si ahora, por ejemplo, ejecutáramos lo siguiente:

```
1 mi_archivo = open('archivo.txt', 'w', encoding='utf-8')
2 mi_archivo.write('Esto es nuevo')
3 mi_archivo.close()
```

Como el archivo `archivo.txt` ya existía, sobrescribimos su contenido, por lo que al abrir el archivo nuevamente, no veremos rastros de lo que habíamos escrito anteriormente:

```
Esto es nuevo
```

Modo append

En este modo, si el archivo no existe, Python lo creará, pero si el archivo ya existe, agregará el contenido en la línea siguiente a la última escrita (no se sobrescribe el archivo). Para escribir archivos con este modo, utilizamos `'a'` en la función `open()` y al igual que con el modo anterior, usamos la función `write()`.

```
1 mi_archivo = open('<Nombre del archivo>', 'a')
2 mi_archivo.write(<String con contenido a escribir>)
3 mi_archivo.close()
```

O su equivalente:

```
1 with open('<Nombre del archivo>', 'a') as mi_archivo:
2     mi_archivo.write(<String con contenido a escribir>)
```

Veamos su aplicación en el siguiente ejemplo:

```
1 mi_archivo = open('archivo.txt', 'a', encoding='utf-8')
2 mi_archivo.write('Escribe esto en el documento\nEsta va a ser otra línea\nY esta otra')
3 mi_archivo.close()
```

Como el archivo `archivo.txt` no existía, estamos creándolo y además le estamos agregando contenido. Si abrimos el archivo, veremos lo siguiente:

```
Escribe esto en el documento
Esta va a ser otra línea
Y esta otra
```

Hasta ahora, nada nuevo con respecto al modo anterior. Pero ejecutemos lo siguiente:

```
1 mi_archivo = open('archivo.txt', 'a', encoding='utf-8')
2 mi_archivo.write('\nEsto es nuevo')
3 mi_archivo.close()
```


Como el archivo `archivo.txt` ya existía, agregamos al final el contenido nuevo, por lo que al abrir el archivo veremos lo que tenía antes, más lo agregado ahora:

```
Escribe esto en el documento
Esta va a ser otra línea
Y esta otra
Esto es nuevo
```

Capítulo 11

Recursión

En este capítulo vamos a explorar la noción de recursión en computación. Esta es una herramienta poderosa que nos permite abordar muchos problemas que de otra forma terminan siendo bastante difíciles.

11.1. Definir una función recursiva

En ciencias de la computación, la recursión es un método de resolución de problemas basado en la capacidad de una función de llamarse a si misma en su definición, con el objetivo de dividir un problema grande, en problemas pequeños. Quizás todos somos familiares a esta noción, ya que la hemos visto muchas veces en contextos matemáticos. Por ejemplo, ¿recuerdas cómo se definía a^n cuando n era un número natural? Exacto, la definición poseía un caso base, además de un caso generalizado:

$$\begin{array}{ll} a^n = 1 & \text{si } n = 0 \\ a^n = a \cdot a^{(n-1)} & \text{si } n > 0 \end{array}$$

Esta misma idea es la que exploraremos ahora. Nosotros podemos definir una función en la que esta se use a si misma. De esta forma, vamos a tener un caso base y un caso generalizado. Un primero ejemplo interesante es escribir una función llamada `eleva_a_n(a, n)` de manera recursiva. Esta función recibe la base a de una potencia y la eleva al exponente n .

```
1 def eleva_a_n(a, n):
2     if n == 0:
3         return a
4     else:
5         return a*eleva_a_n(a, n - 1)
```

Como lo vemos, en el caso de que n sea igual a 0, lo que hacemos es retornar a . En otro caso, vamos a retornar la multiplicación de a por la misma función que acabamos de definir, solo que ahora la estamos llamando con el parámetro $n - 1$. No parece ser muy difícil, ¿cierto?. El concepto en realidad es bastante sencillo, pero hay algunas soluciones que se vuelven más difíciles que otras.

En términos generales una recursión se escribe como:

```
1 def funcion_recursiva(var_1, ..., var_n):
2     if <condición>:
3         # Caso base
4     else:
```

```
5 # Llamada recursiva
```

Veamos un nuevo ejemplo sencillo. Definamos de forma recursiva la función factorial.

Ejemplo 11.1.1. La función factorial de n ($n!$) se define para los naturales como 1 si $n = 0$ y $n(n - 1)!$ en otro caso. Defina la función recursiva en Python que calcule el factorial de un número.

```
1 def factorial(n):
2     if n == 0:
3         return 1
4     else:
5         return n*factorial(n - 1)
```

Aquí el caso base se cumple cuando n es igual a 0. En otro caso pasamos a la llamada recursiva. No es muy distinto a lo que vimos anteriormente.

11.2. Iterar de forma recursiva

Hasta ahora cuando quieres iterar por una lista tus opciones son usar **for** o **while**, sin embargo, también puedes iterar una lista utilizando recursión. Existen paradigmas de programación en donde la recursión se explota ampliamente (por ejemplo, la programación utilizando lógica) y por lo mismo, para iterar, debes recurrir a hacer llamados recursivos. Para entender esta idea, veamos el siguiente ejemplo.

Ejemplo 11.2.1. Define una función cuyo nombre sea `imprimir_lista`, que como parámetro reciba una lista de Python y que imprima cada elemento de la lista en consola. La definición de esta función debe ser recursiva.

```
1 def imprimir_lista(l):
2     if len(l) == 1:
3         print(l[0])
4     else:
5         print(l[0])
6         imprimir_lista(l[1:])
```

Lo que estamos haciendo acá es preguntarnos si el largo de la lista es uno. En ese caso, vamos a imprimir su único elemento. Ahora si la lista tiene largo mayor que uno, lo que vamos a hacer es imprimir su primer elemento, y volver a invocar a la función con la misma lista, pero considerando solo los elementos desde la posición 1 en adelante. Así, por ejemplo si hacemos el llamado:

```
imprimir_lista([1, 2, 3])
```

Lo que va a pasar es que se imprimirá el 1 y luego, en el **else** se va a hacer el llamado a la función:

```
imprimir_lista([2, 3])
```

Luego se imprimirá el 2, y la función se llamará para una lista cuyo único elemento es el 3. Ahí caeremos en el caso base y la recursión terminará.

Ahora, para seguir aprendiendo este concepto, lo mejor es revisar algunos otros ejemplos de funciones recursivas. De esta forma, ahora veremos la permutación de *strings* mediante recursión.

11.3. Permutación de *strings*

El problema de Permutación de *strings* consiste en tomar un string e imprimir todas las posibles permutaciones de los caracteres. Esto es, todos los strings que se pueden formar usando los caracteres del *string* original. Vamos a ver la función que sirve para lograr esto.

```

1 # Funcion que elimina el i-esimo caracter de un string.
2 def eliminar(s, i):
3     if i == 0:
4         return s[1:]
5     elif i == len(s)-1:
6         return s[:len(s)-1]
7     else:
8         return s[:i] + s[i+1:]
9
10 def permutar(s, perm):
11     if len(perm) == 1:
12         # Si el largo del string perm es 1, la recursion termina.
13         print(s + perm)
14     else:
15         for i in range(0, len(perm)):
16             # Toma el caracter y lo pasa
17             s1 = s + perm[i]
18             s2 = eliminar(perm, i)
19             permutar(s1, s2)
20
21 def llamar_permutar(s):
22     permutar('', s)

```

En este problema usamos una estrategia habitual. Se define una función recursiva que recibe dos parámetros, y otra función que sirve para iniciar la recursión. Esta última fija uno de los parámetros de la función recursiva y recibe el input.

La función `permutar` recibe un string ya permutado `s` y otro por permutar `perm`. Dentro del `for` de la función `permutar`, lo que se hace es tomar de a uno los caracteres por permutar, y concatenarlo con el strings ya permutado. Por ejemplo si el input es `abc`, en la primera llamada de `permutar` esta función se llamará recursivamente tres veces:

```

# La llamada de permutar('', 'abc') llama a:
permutar('a', 'bc')
permutar('b', 'ac')
permutar('c', 'ab')

```

Para cada uno de estas llamadas a `permutar` se vuelve a ejecutar el caso recursivo:

```

# La llamada de permutar('a', 'bc') llama a:
permutar('ab', 'c')
permutar('ac', 'b')
# La llamada de permutar('b', 'ac') llama a:
permutar('ba', 'c')
permutar('bc', 'a')
# La llamada de permutar('c', 'ab') llama a:
permutar('ca', 'b')
permutar('cb', 'a')

```

Notamos que todas las llamadas anteriores ejecutan el caso base de la recursión, y cada una imprime respectivamente:

```

abc
acb
bac
bca
cab
cba

```

11.4. Ordenamiento con funciones recursivas

Dos de las más famosas funciones de ordenamiento utilizan recursión. Vamos a darle una mirada ahora.

11.4.1. Quicksort

Quicksort es un ordenamiento recursivo que divide una lista en sublistas y se ordenan de la siguiente forma:

- Elegir un elemento de la lista que se denominará pivote.
- Dejar todos los elementos menores que él a un lado, y todos los mayores que el pivote al otro.
- Obtenemos dos sublistas. Una de todos los elementos de la izquierda, y otra de todos los elementos de la derecha.
- Repetir recursivamente el proceso para cada sublista mientras tenga más de un elemento.

Una forma propuesta de implementar este algoritmo en Python es la siguiente:

```
1 def quicksort(lista):
2     menores = []
3     iguales = []
4     mayores = []
5
6     if len(lista) > 1:
7         pivot = lista[0]
8         for elemento in lista:
9             if elemento < pivot:
10                menores.append(elemento)
11            elif elemento == pivot:
12                iguales.append(elemento)
13            elif elemento > pivot:
14                mayores.append(elemento)
15        return quicksort(menores) + quicksort(iguales) + quicksort(mayores)
16        # El operador + une las tres listas
17        # Ocupamos la misma funcion en cada sublista
18    else:
19        return lista
```

Notamos que `quicksort` es una función que recibe como parámetro una lista. En el inicio se generan tres listas auxiliares que almacenarán los menores, los mayores y los iguales al pivote, que siempre será el primer elemento de cada una de las sublistas. Luego de llenar estas tres listas auxiliares con los elementos correspondientes, debemos volver a llamar a la función `sort` en cada una de ellas. En el ejemplo anterior, debemos notar que `lista` es el parámetro ingresado, por lo que la línea que dice `return lista`, esta retornando los valores de las sublistas menores, iguales o mayores, cuando posean sólo un elemento. Es una buena idea mirar un video que muestre este ordenamiento de manera gráfica, así entenderlo se hace mucho más fácil.

11.4.2. Mergesort

Es un ordenamiento recursivo que se basa en la estrategia "Dividir para conquistar". La idea es dividir una lista en trozos e ir ordenando por segmentos del mismo tamaño. Por ejemplo si queremos ordenar la siguiente lista de menor a mayor:

```
l = [23, 4, 30, 7, 10, 11, 0]
```

La dividimos:

```
[23] [4] [30] [7] [10] [11] [0]
```

Juntamos los pares adyacentes y los ordenamos:

```
[4, 23] [7, 30] [10, 11] [0]
```

Luego tomando los pares de listas adyacentes formamos las siguientes dos listas:

```
[4, 23, 7, 30] [10, 11, 0]
```

Las dos listas están desordenadas, así que las debemos ordenar:

```
[4, 7, 23, 30] [0, 10, 11]
```

Para hacer esto se van tomando los primeros elementos de las sublistas anteriores. Por ejemplo para ordenar este segmento:

```
# Tenemos dos sub-listas
# Que sabemos que por separado están ordenadas:
[4, 23] [7, 30]
```

```
# Creamos una lista resultante vacia:
[4, 23] [7, 30]
[]
```

```
# Pasamos a ella el elemento menor entre 4 y 7:
[23][7,30]
[4]
```

```
# Luego entre 23 y 7:
[23] [30]
[4, 7]
```

```
# Luego entre 23 y 30:
[][30]
[4, 7, 23]
```

```
# Luego el elemento restante:
[]
[4, 7, 23, 30]
```

Nuevamente te recomendamos ver un video que explique *mergesort* visualmente, ya que se te hará mucho más fácil entenderlo. Ahora el código que realiza este procedimiento es el siguiente:

```
1 # Ordenamiento recursivo con mergesort
2 def mergesort(lista):
3     if len(lista) <=1:
4         return lista
5     else:
6         lista_izquierda = []
7         lista_derecha = []
```

```

8     medio = len(lista)//2
9     for i in range(0, medio):
10        lista_izquierda.append(lista[i])
11    for j in range(medio, len(lista)):
12        lista_derecha.append(lista[j])
13
14    lista_izquierda = mergesort(lista_izquierda)
15    lista_derecha = mergesort(lista_derecha)
16
17    # merge es una función definida más abajo
18    # Que nos ayuda a combinar dos sublistas ya ordenadas
19    return merge(lista_izquierda, lista_derecha)
20
21 def merge(lista_izquierda, lista_derecha):
22     lista_retorno = []
23     while len(lista_izquierda) > 0 or len(lista_derecha) > 0:
24
25         if len(lista_izquierda) > 0 and len(lista_derecha) > 0:
26             if lista_izquierda[0] <= lista_derecha[0]:
27                 lista_retorno.append(lista_izquierda[0])
28                 lista_izquierda.pop(0)
29             else:
30                 lista_retorno.append(lista_derecha[0])
31                 lista_derecha.pop(0)
32
33         elif len(lista_izquierda)>0:
34             lista_retorno.append(lista_izquierda[0])
35             lista_izquierda.pop(0)
36
37         elif len(lista_derecha)>0:
38             lista_retorno.append(lista_derecha[0])
39             lista_derecha.pop(0)
40
41     return lista_retorno

```

La función `merge_sort` es recursiva. Va dividiendo en sublistas la lista original para luego ejecutarse a si misma en estas dos sublistas. Este paso se hace hasta dejar las sublistas de tamaño 1. Luego las listas se van juntando nuevamente quedando ordenadas según la función `merge`.

11.5. Ejercicios propuestos

Ejercicio 11.5.1. Algebraco es una aplicación que sirve para manipular expresiones algebraicas. Internamente, Algebraco representa las expresiones aritméticas como:

- un número entero, o
- una lista de tres elementos donde, tanto el primer elemento como el segundo son expresiones, y el tercer elemento es un operador. Los operadores permitidos son los *strings* '+', '-', '*' y '/'.

Así, 4, [1, [3, 4, '+'], '-'] y [[5, 5, '/'], [2, 1, '-'], '+'] son expresiones permitidas en Algebraco que corresponden a 4, $(1-(3+4))$ y $((5/5) + (2-1))$ respectivamente.

Tu misión es escribir una función `evalua`, que tome como argumento una expresión de Algebraco y retorne el número al cual esta expresión es igual. Por ejemplo:

`evalua(4)` debe retornar 4,
`evalua([1, [3, 4, '+'], '-'])` debe retornar -6,
`evalua([[11, 7, '-'], [4, 3, '+'], '*'])` debe retornar 28.

Ayuda: Para saber si la variable `a` es un `int`, puedes preguntar `if type(a) == int`.
Informalmente, las subexpresiones de una expresión `E` son partes de `E` que al mismo tiempo son expresiones.

Por ejemplo, si `E = [1, [2, 3, '-'], '+']`, las subexpresiones de `E` son `[1, [2, 3, '-'], '+']`, `1`, `[2, 3, '-']`, `2` y `3`. Por definición, `E` es siempre una subexpresión de `E`.

Además, debes escribir una función `cuenta` que reciba una expresión y un número, tal que `cuenta(exp, n)` retorne el número de subexpresiones de `exp` que, al evaluarse, tienen el valor `n`. Por ejemplo, `cuenta([1, 1, '+'], 1)` retorna 2 porque hay dos subexpresiones de `[1, 1, '+']`, en particular `1` y `1`, que son iguales a `1`.