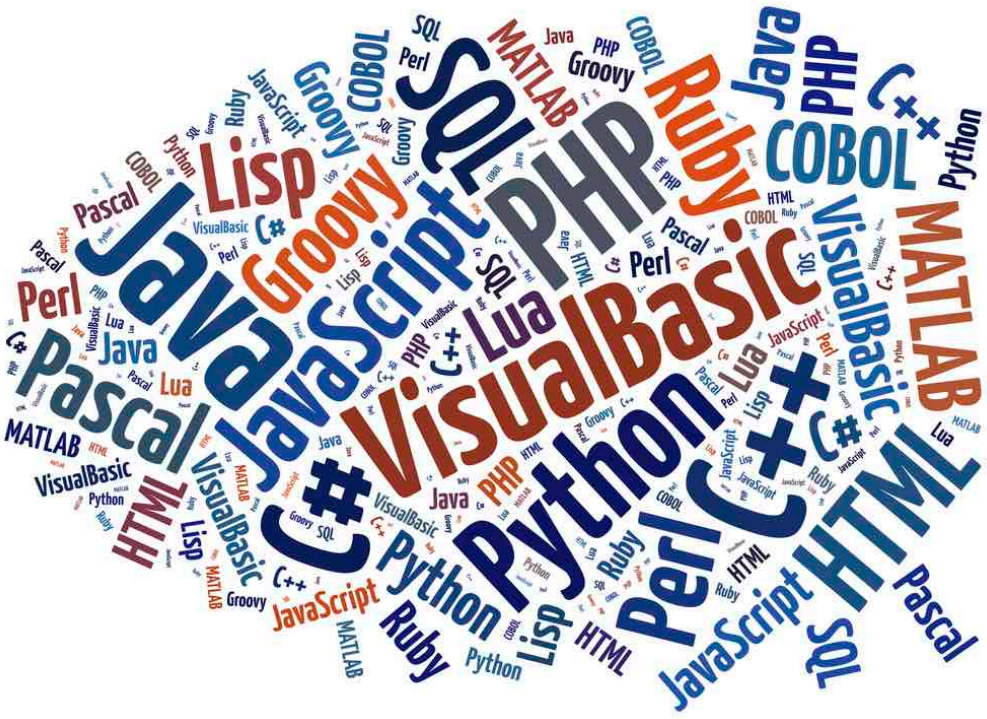


Apuntes para aprender a programar

Introducción a la Programación

Departamento de Ciencias Básicas
Universidad Nacional de Luján

Contenidos

Unidad 1

Algunos conceptos básicos

En esta unidad hablaremos de lo que es un programa de computadora e introduciremos unos cuantos conceptos referidos a la programación y a la ejecución de programas. Utilizaremos en todo momento el lenguaje de programación Python para ilustrar esos conceptos.

1.1 Conceptos iniciales

En la actualidad, la mayoría de nosotros utilizamos computadoras permanentemente: para mandar correos electrónicos, navegar por Internet, chatear, jugar, escribir textos.

Las computadoras se usan para actividades tan disímiles como predecir las condiciones meteorológicas de la próxima semana, guardar historias clínicas, diseñar aviones, llevar la contabilidad de las empresas o controlar una fábrica. Y lo interesante aquí (y lo que hace apasionante a esta carrera) es que el mismo aparato sirve para realizar todas estas actividades: uno no cambia de computadora cuando se cansa de chatear y quiere jugar al solitario.

1.1.1 La informática y la computadora

Una de las definiciones que podríamos aceptar de la palabra ciencia es aquella que postula que esta es una rama del saber humano constituida por el conjunto de conocimientos objetivos y verificables sobre una materia determinada que es utilizada para la resolución de problemas.

En este sentido, y de acuerdo a la Real Académia Española, la Informática es un "conjunto de conocimientos científicos y técnicas que hacen posible el tratamiento automático de la información por medio de ordenadores."

Nosotros normalmente utilizaremos el término computadoras y no ordenadores. Por su parte, una computadora está formada, principalmente, por dos tipos de componentes:

- Hardware: Conjunto de elementos físicos relacionados que componen una computadora,
- Software: Son los datos y programas que hacen funcionar a una computadora y es de carácter intangible.

A su vez, muchos definen una computadora moderna como "una máquina que almacena y manipula información bajo el control de un programa que puede cambiar". Aparecen acá dos conceptos que son claves: por un lado se habla de una *máquina* que almacena información, y por el otro lado, esta máquina está controlada por *un programa que puede cambiar*.

Una calculadora sencilla, de esas que sólo tienen 10 teclas para los dígitos, una tecla para cada una de las 4 operaciones, un signo igual, encendido y CLEAR, también es una máquina

que almacena información y que está controlada por un programa. Pero lo que diferencia a esta calculadora de una computadora es que en la calculadora el programa no puede cambiar. Ahora bien, ¿Qué es un programa?

1.1.2 Conceptos de programa y algoritmo

Un *programa de computadora* es una secuencia de *instrucciones* definidas paso a paso (como veremos más adelante, un algoritmo) que le indican a una computadora cómo realizar una tarea dada. En la computadora uno puede modificar un programa de acuerdo a la tarea que quiere realizar.

Las instrucciones se deben escribir en un lenguaje que nuestra computadora entienda. Los lenguajes de programación son lenguajes diseñados para dar órdenes a una computadora, de manera exacta y no ambigua. Sería muy agradable poder darle las órdenes a la computadora en castellano, pero el problema del castellano, y de las lenguas habladas en general, es su ambigüedad. Por ejemplo, si alguien nos dice *"Comprá el collar sin monedas"*, no sabremos si nos pide que compremos el collar que no tiene monedas, o que compremos un collar y que no usemos monedas para ello. Tales dudas no pueden aparecer cuando se le dan órdenes a una computadora.

Este curso va a tratar precisamente de cómo se escriben programas para hacer que una computadora realice una determinada tarea. Vamos a usar un lenguaje específico, Python, porque es sencillo y elegante, pero éste no será un curso de Python sino un curso de programación.

Sabías que...

Existen cientos de lenguajes de programación, y Python es uno de los más utilizados en la industria del software. Entre sus usos más frecuentes se destacan las aplicaciones web, computación científica e inteligencia artificial. Muchas empresas hacen extensivo uso de Python, entre ellas gigantes como **Google**, **NASA**, **Facebook** y **Amazon**. Python también suele ser incluido como herramienta de *scripting* embebido en ciertos paquetes de software, por ejemplo en programas de modelado y animación 3D como **3ds Max** y **Blender**, o videojuegos como **Civilization IV**.

Otra definición aceptada es que un programa es un algoritmo escrito a partir de un lenguaje de programación. Hasta hace no mucho tiempo se utilizaba el término algoritmo para referirse únicamente a formas de realizar ciertos cálculos, pero con el surgimiento de la computación, el término algoritmo pasó a abarcar cualquier método para obtener un resultado.

Sabías que...

La palabra *algoritmo* proviene de *algorismo*. En la antigüedad, los *algoristas* eran los que calculaban usando la numeración arábiga y mientras que los *abacistas* eran los que calculaban usando ábacos. Con el tiempo el *algorismo* se deformó en *algoritmo*, influenciado por el término *aritmética*.

A su vez, el uso de la palabra *algorismo* proviene del nombre de un matemático persa famoso, en su época y para los estudiosos de esa época, Abu Abdallah Muhammad ibn Mûsâ al-Jwârizmî. Al-Juarismi, como se lo llama usualmente, escribió en el año 825 el libro "Al-Kitâb al-mukhtasar fî hîsâb al-gabr wa'l-muqâbala" (Compendio del cálculo por el método de completado y balanceado), del cual surgió también la palabra "álgebra".

En cuanto a su definición formal[1], se entiende por algoritmo a:

> Algoritmo es la especificación rigurosa de la secuencia de pasos (instrucciones) a realizar sobre un autómata para alcanzar un resultado deseado en tiempo finito.

El abordar toda la definición en su conjunto puede resultar compleja pero la descompondremos, al igual que hicimos antes, para entenderla:

- En primer lugar, vemos que es "una especificación rigurosa"; esto quiere decir que las instrucciones no deben ser ambiguas, y debe entenderse perfectamente, y de forma unívoca.
- Luego luce "a realizar sobre un autómata"; cuando programemos nuestro autómata será una computadora en primer término y otros dispositivos en el futuro pero no necesariamente un autómata es una computadora. Incluso podríamos escribir un algoritmo para que un compañero de curso (autómata) pueda llegar a nuestro hogar.
- La definición finaliza diciendo "para alcanzar un resultado deseado en tiempo finito". Esta frase grafica dos cuestiones: la primera, y mas intuitiva, es que todos los algoritmos tienen un resultado deseado, un objetivo buscado; la segunda, que hace alusión al "tiempo finito" supone que el algoritmo, tanto en su definición como en su ejecución, deben tener un inicio y un fin. Veremos mas adelante que los algoritmos modernos pueden poner en discusión esta última característica pero por ahora la aceptaremos.

Es importante prestar especial atención en el concepto abstracto de algoritmo dado que en programación, el análisis del problema y la construcción del algoritmo son etapas claves en el proceso de creación de un programa informático. A partir del algoritmo terminado, solo nos resta poder "traducirlo" a un lenguaje de programación que entienda nuestro autómata, la computadora.

1.2 Lenguajes de programación

La computadora entiende únicamente un lenguaje llamado lenguaje de máquina o código máquina. El código máquina es muy simple y francamente muy pesado de escribir, ya que está representado en su totalidad por solamente ceros y unos:

```
00101000111010010010101000001111
11100110000011101010010101011010
...
```

Entonces, el lenguaje de programación será nuestro intermediario entre la computadora y nosotros. Es el instrumento a través del cual nosotros podremos dar instrucciones a una máquina para que ésta las ejecute. En nuestro caso utilizaremos Python, un lenguaje ampliamente utilizado y muy sencillo.

Para que una computadora pueda comprender las instrucciones que contiene un programa, el código fuente debe estar escrito en un lenguaje de programación, el cual debe convertirse a un formato legible por una máquina.

A partir de como se desarrolle este procedimiento, desde el punto de vista conceptual, podemos dividir a los lenguajes de programación en dos grandes grupos:

[1] DE GIUSTI, A. & OTROS (2001). "Algoritmos, datos y programas". Prentice Hall.

Sabías que...

Python fue creado a finales de los años 80 por un programador holandés llamado Guido van Rossum, quien se desempeñó como líder del desarrollo del lenguaje hasta 2018.

La versión 2.0, lanzada en 2000, fue un paso muy importante para el lenguaje ya que era mucho más madura, incluyendo un *recolector de basura*. La versión 2.2, lanzada en diciembre de 2001, fue también un hito importante ya que mejoró la orientación a objetos. La última versión de esta línea es la 2.7 que fue lanzada en noviembre de 2010 y estará vigente hasta 2020.

En diciembre de 2008 se lanzó la rama 3.0 (en este libro utilizamos la versión 3.7). Python 3 fue diseñado para corregir algunos defectos de diseño en el lenguaje, y muchos de los cambios introducidos son incompatibles con las versiones anteriores. Por esta razón, las ramas 2.x y 3.x coexisten con distintos grados de adopción.

- Lenguajes interpretados: cuentan con un intérprete, que es un programa informático que procesa el código fuente durante su tiempo de ejecución, es decir, mientras el software se está ejecutando, y actúa como una interfaz entre el programa y la computadora. El intérprete siempre procesa el código línea por línea, de modo que lee, analiza y prepara cada secuencia de forma consecutiva para el procesador.
- Lenguajes compilados: Estos lenguajes cuentan con un compilador, el cual consiste en un programa que traduce todo el código fuente a código de máquina antes de ejecutarlo. Recién luego de este proceso el procesador ejecuta el software, obteniendo todas las instrucciones en código de máquina antes de comenzar.

Con algunos reparos tecnológicos que escapan al alcance de este curso, Python se considera un lenguaje interpretado.

1.3 Cómo darle instrucciones a la máquina usando Python

El lenguaje Python nos provee de un *intérprete*, es decir un programa que interpreta las órdenes que le damos a medida que las escribimos. La forma más típica de invocar al intérprete es ejecutar el comando `python3` en la **terminal**.

Atención

De forma tal de aprovechar al máximo este libro, recomendamos instalar Python 3 en una computadora, y acompañar la lectura probando todos los ejemplos de código y haciendo los ejercicios.

En `https://www.python.org/downloads/` se encuentran los enlaces para descargar Python, y en `http://docs.python.org.ar/tutorial/3/interpreter.html` hay más información acerca de cómo ejecutar el intérprete en cada sistema operativo.

1.3.1 La terminal o consola

La *terminal* del sistema operativo permite ingresar órdenes a la computadora en forma de líneas de texto. Los tres sistemas operativos más populares (Windows, Mac OS y Linux) están equipados con una terminal. Está fuera del alcance de este apunte cubrir el uso detallado de la terminal, pero para empezar será suficiente con saber cómo acceder a la misma:

- En Windows, presionar las teclas Windows + R, luego escribir `cmd` y presionar Enter.

- En Mac OS, presionar las teclas ⌘ + Espacio, luego escribir `terminal` y presionar Enter.
- En Linux (Ubuntu), presionar Ctrl + Alt + T.

La terminal debería mostrar algo como se ve en la Figura 1.1. En la figura se muestra la terminal en un sistema operativo Linux; en otros sistemas operativos puede verse ligeramente diferente, pero siempre debería mostrar un espacio de texto con un cursor para escribir.

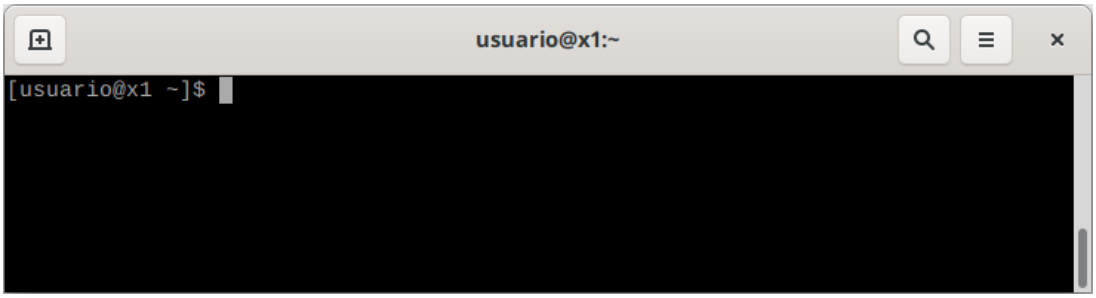


Figura 1.1: La terminal en un sistema operativo Linux.

1.3.2 El intérprete interactivo de Python

Una vez que accedimos a la terminal del sistema operativo, el próximo paso es abrir el intérprete de Python. Para eso, escribimos `python3` y presionamos Enter (Figura 1.2).

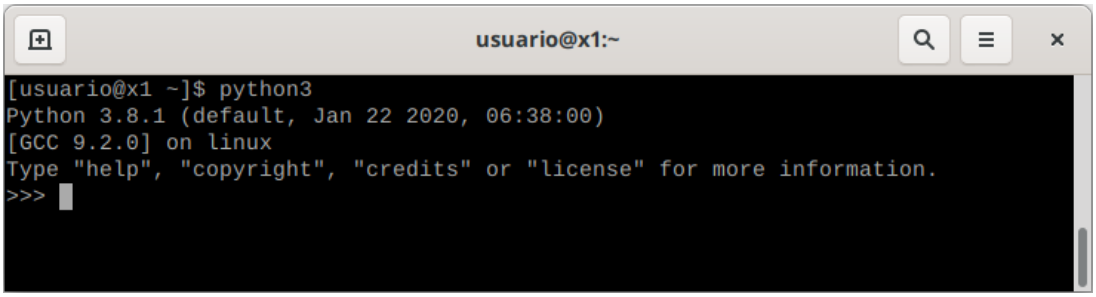


Figura 1.2: El intérprete de Python.

A partir de ahora mostraremos el contenido de la terminal utilizando el siguiente formato:

```
$ python3 ❶
Python 3.6.0 (default, Dec 23 2016, 11:28:25)
[GCC 6.2.1 20160830] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> ❷
```

❶ Las líneas que comienzan con `$` indican órdenes que le damos al sistema operativo (en este caso la orden es `python3`, es decir *abrir el intérprete de Python*).

❷ Para orientarnos, el intérprete de Python muestra los símbolos `>>>` (llamaremos a esto el *prompt*), indicando que podemos escribir a continuación una *sentencia* u orden que será evaluada por Python (en lugar de ser evaluada directamente por el sistema operativo).

Algunas sentencias sencillas, por ejemplo, permiten utilizar el intérprete como una calculadora simple con números enteros. Para esto escribimos la *expresión* que queremos resolver luego

del *prompt* y presionamos la tecla Enter. El intérprete de Python evalúa la expresión y muestra el resultado en la línea siguiente. Luego nos presenta nuevamente el *prompt*.

```
>>> 2+3
5
>>>
```

Python permite utilizar las operaciones `+`, `-`, `*`, `/`, `//` y `**` (suma, resta, multiplicación, división, división entera y potencia). La sintaxis es la convencional (valores intercalados con operaciones), y se puede usar paréntesis para modificar el orden de asociación natural de las operaciones (potencia, producto/división, suma/resta).

```
>>> 5*7
35
>>> 2+3*7
23
>>> (2+3)*7
35
>>> 10/4
2.5
>>> 10//4
2
>>> 5**2
25
```

1.4 Valores y tipos

En la operación `5 * 7` cuyo resultado es `35`, decimos que `5`, `7` y `35` son *valores*. En Python, cada valor tiene un *tipo de dato* asociado. El tipo de dato del valor `35` es *número entero*.

Hay dos tipos de datos numéricos: los **números enteros** y los **números de punto flotante**. Los números enteros (`42`, `0`, `-5`, `10000`) representan el valor entero exacto que ingresemos. Los números de punto flotante (`5.3`, `-98.28109`, `0.0`)[2] son parecidos a la notación científica, almacenan una cantidad limitada de dígitos significativos y un exponente, por lo que sirven para representar magnitudes en forma aproximada. Según los operandos y las operaciones que hagamos usaremos la aritmética de los enteros o de los de punto flotante.

Vamos a elegir enteros cada vez que necesitemos recordar un valor exacto: la cantidad de alumnos, cuántas veces repito una operación, un número de documento, la cantidad de tornillos producidos.

Cuando operamos con números enteros, el resultado es exacto:

```
>>> 1 + 2
3
```

Vamos a elegir punto flotante cuando nos interese más la magnitud y no tanto la exactitud, lo cual suele ser típico en la física y la ingeniería: la temperatura, el seno de un ángulo, la distancia recorrida, la altura de una persona expresada en metros.

Cuando hay números de punto flotante involucrados en la operación, el resultado es aproximado:

```
>>> 0.1 + 0.2
0.30000000000000004
```

[2]Notar que se utiliza el punto decimal y no la coma decimal.

Además de efectuar operaciones matemáticas, Python nos permite trabajar con porciones de texto, que llamaremos **cadenas**, y que se introducen entre comillas simples (') o dobles (").

```
>>> '¡Hola Mundo!'
'¡Hola Mundo!'
>>> 'abcd' + 'efgh'
'abcdefgh'
>>> 'abcd' * 3
'abcdabcdabcd'
```

1.5 Variables

Python nos permite asignarle un nombre a un valor, de forma tal de "recordarlo" para usarlo posteriormente, mediante la sentencia `<nombre> = <expresión>`.

```
>>> x = 8
>>> x
8
>>> y = x * x
>>> 2 * y
128
>>> lenguaje = 'Python'
>>> 'Estoy programando en ' + lenguaje
'Estoy programando en Python'
```

En este ejemplo creamos tres *variables*, llamadas `x`, `y` y `lenguaje`, y las asociamos a los valores 8, 64 y `'Python'`, respectivamente. Luego podemos usar esas variables como parte de cualquier expresión, y en el momento de evaluarla, Python reemplazará las variables por su valor asociado.

1.6 Funciones

Para efectuar algunas operaciones particulares necesitamos introducir el concepto de *función*:

```
>>> abs(10)
10
>>> abs(-10)
10
>>> max(5, 9, -3)
9
>>> min(5, 9, -3)
-3
>>> len("abcd")
4
```

Una función es un fragmento de programa que permite efectuar una operación determinada. `abs`, `max`, `min` y `len` son ejemplos de funciones de Python: la función `abs` permite calcular el valor absoluto de un número, `max` y `min` permiten obtener el máximo y el mínimo entre un conjunto de números, y `len` permite obtener la longitud de una cadena de texto.

Una función puede recibir 0 o más *parámetros* o *argumentos* (expresados entre entre paréntesis, y separados por comas), efectúa una operación y devuelve un *resultado*. Por ejemplo, la función `abs` recibe un parámetro (un número) y su resultado es el valor absoluto del número.

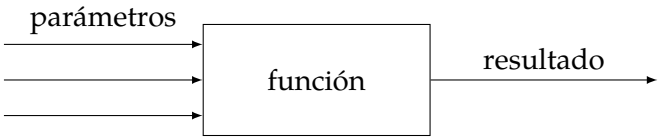


Figura 1.3: Una función recibe parámetros y devuelve un resultado.

Python viene equipado con muchas funciones, pero ya hemos dicho que, como programadores, debíamos ser capaces de escribir nuevas instrucciones para la computadora. Los programas de correo electrónico, navegación web, chat, juegos, procesamiento de texto o predicción de las condiciones meteorológicas de los próximos días no son más que grandes programas implementados introduciendo nuevas funciones a la máquina, escritas por uno o muchos programadores.

1.7 Construir programas y módulos

El intérprete interactivo es muy útil para probar cosas, acceder a la ayuda, inspeccionar el lenguaje, etc, pero tiene una gran limitación: ¡cuando cerramos el intérprete perdemos todas las definiciones! Para conservar los programas que vamos escribiendo, debemos escribir el código utilizando algún editor de texto, y guardar el archivo con la extensión `.py`.

Sabías que...

El intérprete interactivo de python nos provee una ayuda en línea; es decir, nos puede dar la documentación de cualquier función o instrucción. Para obtenerla llamamos a la función `help()`. Si le pasamos por parámetro el nombre de una función (por ejemplo `help(abs)` o `help(range)`) nos dará la documentación de esa función. Para obtener la documentación de una instrucción la debemos poner entre comillas; por ejemplo: `help('for')`, `help('return')`.

En el código 1.1 se muestra nuestro primer programa, `cuad2.py`, que nos permite calcular la suma de los cuadrados de dos números.

Código 1.1 `cuad2.py`**:** Imprime la suma de los cuadrados de dos números

```
1 n = 2
2 m = 3
3 print("La suma de los cuadrados de ", n, " y ", m, " es ", n*n+m*m)
```

En la última línea del programa introducimos una función nueva: `print()`. La función `print` recibe uno o más parámetros de cualquier tipo y los imprime en la pantalla. ¿Por qué no habíamos utilizado `print` hasta ahora?

En el modo interactivo, Python imprime el resultado de cada expresión luego de evaluarla:

```
>>> 2 + 2
4
>>> "Hola" + "Mundo"
HolaMundo
```

```
>>> 11 * 8
88
```

En cambio, cuando Python ejecuta un programa `.py` no imprime absolutamente nada en la pantalla, a menos que le indiquemos explícitamente que lo haga. Por eso es que en `cuad2.py` debemos llamar a la función `print` para mostrar el resultado.

Para ejecutar el programa debemos abrir una consola del sistema y ejecutar `python cuad2.py`:

```
$ python3 cuad2.py
La suma de los cuadrados de 2 y 3 es 13
```

1.8 Interacción con el usuario

Ya vimos que la función `print` nos permite mostrar información al usuario del programa. En algunos casos también necesitaremos que el usuario ingrese datos al programa, para esto podemos usar la función `input`.

Problema 1.8.1. Escribir en Python un programa que pida al usuario que escriba su nombre, y luego lo salude.

Código 1.2 `saludar.py`**:** Saluda al usuario posr su nombre

```
1 nombre = input("Por favor ingrese su nombre: ")
2 saludo = "Hola " + nombre + "!"
3 print(saludo)
```

En el Código 1.2 usamos la función `input` para pedirle al usuario su nombre. `input` presenta al usuario el mensaje que le pasamos por parámetro, y luego le permite ingresar una cadena de texto. Cuando el usuario presiona la tecla Enter, `input` devuelve la cadena ingresada. Luego concatenamos cadenas de caracteres para generar el saludo, y llamamos a `print` para mostrar el saludo al usuario.

Para ejecutar el programa, nuevamente escribimos en la consola del sistema:

```
$ python3 saludar.py
Por favor ingrese su nombre: Alan
Hola Alan!
```

Problema 1.8.2. Escribir en Python un programa que haga lo siguiente:

1. Muestra un mensaje de bienvenida por pantalla.
2. Le pide al usuario que introduzca dos números enteros $n1$ y $n2$.
3. Imprime la suma de los cuadrados de los dos números enteros introducidos.
4. Muestra un mensaje de despedida por pantalla.

Solución. La solución a este problema se encuentra en el Código 1.3.

Como siempre, podemos ejecutar el programa en la consola del sistema:

Código 1.3 `suma_cuadrados.py`**:** Imprime los cuadrados solicitados

```
print("Se calculará la suma de los cuadrados")

n1 = int(input("Ingrese un número entero: "))
n2 = int(input("Ingrese otro número entero: "))
suma = 0
suma = suma + n1*n1
suma = suma + n2*n2

print(suma)
print("Es todo por ahora")
```

```
$ python3 suma_cuadrados.py
Se calculará la suma de los cuadrados
Ingrese un número entero: 5
Ingrese otro número entero: 8
61
Es todo por ahora
```

En el Código 1.3 aparece una función que no habíamos utilizado hasta ahora: `int`. ¿Por qué es necesario utilizar `int` para resolver el problema?

En un programa Python podemos operar con cadenas de texto o con números. Las representaciones dentro de la computadora de un número y una cadena son distintas. Por ejemplo, los números `0`, `42` y `12345678` se almacenan como números binarios ocupando todos la misma cantidad de memoria (4 u 8 bytes), mientras que las cadenas `"0"`, `"42"` y `"12345678"` son secuencias de caracteres, en las que cada dígito se representa como un caracter y cada caracter ocupa típicamente 1 byte.

La función `input` interpreta cualquier valor que el usuario ingresa mediante el teclado como una cadena de caracteres. Es decir, `input` siempre devuelve una cadena, incluso aunque el usuario haya ingresado una secuencia de dígitos.

Por eso es que introducimos la función `int`, que devuelve el parámetro que recibe *convertido* a un número entero:

```
>>> int("42")
42
```

1.9 Estado y computación

A lo largo de la ejecución de un programa las variables pueden cambiar el valor con el que están asociadas. En un momento dado uno puede detenerse a observar a qué valor se refiere cada una de las variables del programa. Esa "foto" que indica en un momento dado a qué valor hace referencia cada una de las variables se denomina *estado*. También hablaremos del *estado de una variable* para indicar a qué valor está asociada esa variable, y usaremos la notación `n` → 13 para describir el estado de la variable `n` (e indicar que está asociada al número 13).

A medida que las variables cambian de valores a los que se refieren, el programa va cambiando de estado. La sucesión de todos los estados por los que pasa el programa en una ejecución dada se denomina *computación*.

Para ejemplificar estos conceptos veamos qué sucede cuando se ejecuta el programa `suma\_cuadrados.py`:

Instrucción	Qué sucede	Estado
`print("Se calculará la suma de los cuadrados")`	Se despliega el texto "Se calculará la suma de los cuadrados" en la pantalla.	
`n1 = int(input("Ingrese un número entero: "))`	Se despliega el texto "Ingrese un número entero: " en la pantalla y el programa se queda esperando que el usuario ingrese un número.	
	Supondremos que el usuario ingresa el número 3 y luego oprime la tecla Enter. Se asocia el número 3 con la variable `n1`.	`n1` → 3
`n2 = int(input("Ingrese otro número entero: "))`	Se despliega el texto "Ingrese otro número entero:" en la pantalla y el programa se queda esperando que el usuario ingrese un número.	`n1` → 3
	Supondremos que el usuario ingresa el número 5 y luego oprime la tecla Enter. Se asocia el número 5 con la variable `n2`.	`n1` → 3 `n2` → 5
`suma = 0`	Se asocia el número 0 con la variable `suma`.	`n1` → 3 `n2` → 5 `suma` → 0
`suma = suma + n1*n1`	Se asocia a la variable `suma`, la suma del cuadrado del valor de la variable n1 con el valor de la variable suma.	`n1` → 3 `n2` → 5 `suma` → 9
`suma = suma + n2*n2`	Se asocia a la variable `suma`, la suma del cuadrado del valor de la variable n2 con el valor de la variable suma.	`n1` → 3 `n2` → 5 `suma` → 36
`print(suma)`	Se imprime por pantalla el valor de `suma` (36)	`n1` → 3 `n2` → 5 `suma` → 36
`print("Es todo por ahora")`	Se despliega por pantalla el mensaje "Es todo por ahora"	`n1` → 3 `n2` → 5 `suma` → 36

1.9.1 Depuración de programas

Una manera de seguir la evolución del estado es insertar instrucciones de impresión en sitios críticos del programa. Esto nos será de utilidad para detectar errores y también para comprender cómo funcionan determinadas instrucciones.

Por ejemplo, podemos insertar llamadas a la función `print` en el Código 1.3 para inspeccionar el contenido de las variables:

```
print("Se calculará la suma de los cuadrados")

n1 = int(input("Ingrese un número entero: "))
print("el valor de n1 es:", n1)
n2 = int(input("Ingrese otro número entero: "))
print("el valor de n2 es:", n2)

suma = 0
print("el valor de suma es:", suma)
suma = suma + n1*n1
print("el valor de suma es:", suma)
suma = suma + n2*n2
print("el valor de suma es:", suma)

print(suma)
print("Es todo por ahora")
```

En este caso, la salida del programa será:

```
$ python3 suma_cuadrados.py
Se calculará la suma de los cuadrados
Ingrese un número entero: 5
el valor de n1 es: 5
Ingrese otro número entero: 8
el valor de n2 es: 8
el valor de suma es: 0
el valor de suma es: 25
el valor de suma es: 61
61
Es todo por ahora
```

Si utilizamos este método para depurar el programa, tendremos que recordar eliminar las llamadas `print` una vez que terminemos de escribir y depurar nuestro código.

Unidad 2

Programas sencillos

En esta unidad presentamos los elementos básicos de un programa y empezamos a resolver problemas sencillos, y a programarlos en Python.

A su vez, iniciamos la unidad brindando una metodología inicial recomendada para creación de programas.

2.1 Construcción de programas

Cuando nos disponemos a escribir un programa debemos seguir una cierta cantidad de pasos para asegurarnos de que tendremos éxito en la tarea. La acción irreflexiva (me siento frente a la computadora y escribo rápidamente y sin pensar lo que me parece que es la solución) no constituye una actitud profesional (e ingenieril) de resolución de problemas. Toda construcción tiene que seguir una metodología, un protocolo de desarrollo.

Existen muchas metodologías para construir programas, pero en este curso aplicaremos una sencilla, que es adecuada para la construcción de programas pequeños, y que se puede resumir en los siguientes pasos:

1. **Analizar el problema.** Entender profundamente *cuál* es el problema que se trata de resolver, incluyendo el contexto en el cual se usará.

   > Una vez analizado el problema, asentar el análisis por escrito.

2. **Especificar la solución.** Éste es el punto en el cual se describe *qué* debe hacer el programa, sin importar el cómo. En el caso de los problemas sencillos que abordaremos, deberemos decidir cuáles son los datos de entrada que se nos proveen, cuáles son las salidas que debemos producir, y cuál es la relación entre todos ellos.

   > Al especificar el problema a resolver, documentar la especificación por escrito.

3. **Diseñar la solución.** Éste es el punto en el cuál atacamos el *cómo* vamos a resolver el problema, cuáles son los algoritmos y las estructuras de datos que usaremos. Analizamos posibles variantes, y las decisiones las tomamos usando como dato de la realidad el contexto en el que se aplicará la solución, y los costos asociados a cada diseño.

Luego de diseñar la solución, asentar por escrito el diseño, asegurándonos de que esté completo.

4. **Implementar el diseño.** Traducir a un lenguaje de programación (en nuestro caso, y por el momento, Python) el diseño que elegimos en el punto anterior.

   La implementación también se debe documentar, con comentarios dentro y fuera del código, al respecto de qué hace el programa, cómo lo hace y por qué lo hace de esa forma.

5. **Probar el programa.** Diseñar un conjunto de pruebas para probar cada una de sus partes por separado, y también la correcta integración entre ellas. Utilizar la *depuración* como instrumento para descubir dónde se producen ciertos errores.

   Al ejecutar las pruebas, documentar los resultados obtenidos.

6. **Mantener el programa.** Realizar los cambios en respuesta a nuevas demandas.

   Cuando se realicen cambios, es necesario documentar el análisis, la especificación, el diseño, la implementación y las pruebas que surjan para llevar estos cambios a cabo.

2.2 La programación estructurada y sus estructuras de control

Hasta aquí, venimos abordando algunos conceptos básicos de programación y también mencionamos al pasar que cada paradigma y cada lenguaje de programación tienen un conjunto mínimo de instrucciones con los cuales trabaja.

Está demostrado que un lenguaje de programación con solo tres instrucciones: asignación, decisión e iteración, permite escribir cualquier algoritmo.

Una visión interesante de un programa informático es la que plantea De Giusti: Un programa corresponde a una transformación de datos. A partir de un contexto determinado por los datos ingresados e iniciales (pre-condiciones), el programa transforma la información y debería llegar al resultado esperado produciendo un nuevo contexto (pos-condiciones).

Mediante la programación estructurada todas las bifurcaciones de control de un programa se encuentran estandarizadas, de forma tal que es posible leer los programas desde su inicio hasta su final en forma continua, sin tener que saltar de un lugar a otro del programa siguiendo el rastro de la lógica establecida por el programador.

Como ya adelantamos, los lenguajes de programación tienen un conjunto mínimo de instrucciones que permiten especificar el control del algoritmo a implementar. Este conjunto de instrucciones se denominan estructuras de control.

Las estructuras de control básicas (y que iremos viendo a lo largo de este apunte) son las siguientes:

- **Secuencia:** Sucesión de dos o más operaciones cuya ejecución coincide con el orden de aparición de las instrucciones.
- **Decisión:** Bifurcación entre dos alternativas condicionada por los datos del problema. La instrucción evalúa una condición y ejecuta el conjunto de instrucciones que corresponda al valor de verdad retornado.

- **Iteración:** Repetición de un conjunto de instrucciones mientras se cumpla una condición.

Estos tres tipos de estructuras lógicas de control pueden ser combinadas para producir programas que manejen cualquier tarea de procesamiento de información.

La programación Estructurada esta basada en el Teorema de la Estructura (1966, Böhm - Jiacopini), el cual establece que cualquier programa propio es equivalente a un programa que contiene solamente las estructuras lógicas mencionadas anteriormente.

Sabías que...

Un programa se define como propio si cumple con los requerimientos siguientes:

- Tiene exactamente una entrada y una salida para control del programa.
- Existen caminos seguibles desde la entrada hasta la salida que conducen por cada parte del programa, es decir, no existen lazos infinitos ni instrucciones que no se ejecutan.
- No contiene ciclos infinitos.

Un programa estructurado está compuesto de segmentos o bloques, los cuales pueden estar constituidos por unas pocas instrucciones o por una página o más de codificación. Cuando varios programas propios se combinan, utilizando las tres estructuras básicas de control, el resultado es también un programa propio.

A su vez, un programa propio puede descomponerse en varios más chicos, pero también propios, esta técnica se conoce como Modularización o diseño Top-Down.

2.3 Realizando un programa sencillo

Ahora, pongamos manos a la obra: **¡Vamos a iniciar con el diseño y desarrollo de nuestro primer programa en Python!**

Al leer un artículo en una revista que contiene información de longitudes expresadas en millas, pies y pulgadas, queremos poder convertir esas distancias de modo que sean fáciles de entender. Para ello, decidimos escribir un programa que convierta las longitudes del sistema inglés al sistema métrico decimal.

Antes de comenzar a programar, utilizamos la guía de la sección anterior, para analizar, especificar, diseñar, implementar y probar el problema.

1. **Análisis del problema.** En este caso el problema es sencillo: nos dan un valor expresado en millas, pies y pulgadas y queremos transformarlo en un valor en el sistema métrico decimal. Sin embargo hay varias respuestas posibles, porque no hemos fijado en qué unidad queremos el resultado. Supongamos que decidimos que queremos expresar todo en metros.

2. **Especificación.** Debemos establecer la relación entre los datos de entrada y los datos de salida. Ante todo debemos averiguar los valores para la conversión de las unidades básicas. Buscando en Internet encontramos la siguiente tabla:

   - 1 milla = 1.609344 km
   - 1 pie = 30.48 cm
   - 1 pulgada = 2.54 cm

> **Atención**
>
> A lo largo de todo el curso usaremos punto decimal, en lugar de coma decimal, para representar valores no enteros, dado que esa es la notación que utiliza Python.

La tabla obtenida no traduce las longitudes a metros. La manipulamos para llevar todo a metros:

- 1 milla = 1609.344 m
- 1 pie = 0.3048 m
- 1 pulgada = 0.0254 m

Si una longitud se expresa como L millas, F pies y P pulgadas, su conversión a metros se calculará como:

$$M = 1609.344 * L + 0.3048 * F + 0.0254 * P$$

Hemos especificado el problema. Pasamos entonces a la próxima etapa.

3. **Diseño.** La estructura de este programa es sencilla: leer los datos de entrada, calcular la solución, mostrar el resultado, o *Entrada-Cálculo-Salida*.

   Antes de escribir el programa, escribiremos en *pseudocódigo* (un castellano preciso que se usa para describir lo que hace un programa) una descripción del mismo:

```
Leer cuántas millas tiene la longitud dada
 (y referenciarlo con la variable millas)

Leer cuántos pies tiene la longitud dada
 (y referenciarlo con la variable pies)

Leer cuántas pulgadas tiene la longitud dada
 (y referenciarlo con la variable pulgadas)

Calcular metros = 1609.344 * millas +
    0.3048 * pies + 0.0254 * pulgadas

Mostrar por pantalla la variable metros
```

4. **Implementación.** Ahora estamos en condiciones de traducir este pseudocódigo a un programa en lenguaje Python:

5. **Prueba.** Probaremos el programa con valores para los que conocemos la solución:

   - 1 milla, 0 pies, 0 pulgadas (el resultado debe ser 1609.344 metros).
   - 0 millas, 1 pie, 0 pulgada (el resultado debe ser 0.3048 metros).
   - 0 millas, 0 pies, 1 pulgada (el resultado debe ser 0.0254 metros).

   La prueba la documentaremos con la sesión de Python correspondiente a las tres invocaciones a `ametrico.py`.

En la sección anterior hicimos hincapié en la necesidad de documentar todo el proceso de desarrollo. En este ejemplo la documentación completa del proceso lo constituye todo lo escrito en esta sección.

Código 2.1 `ametrico.py`**:** Convierte medidas inglesas a sistema metrico

```
print("Convierte medidas inglesas a sistema metrico")

millas = int(input("Cuántas millas?: "))
pies = int(input("Y cuántos pies?: "))
pulgadas = int(input("Y cuántas pulgadas?: "))

metros = 1609.344 * millas + 0.3048 * pies + 0.0254 * pulgadas
print("La longitud es de ", metros, " metros")
```

2.4 Piezas de un programa Python

Cuando empezamos a hablar en un idioma extranjero es posible que nos entiendan pese a que cometamos errores. No sucede lo mismo con los lenguajes de programación: la computadora no nos entenderá si nos desviamos un poco de alguna de las reglas.

Por eso es que para poder empezar a programar en Python es necesario conocer los elementos que constituyen un programa en dicho lenguaje y las reglas para construirlos.

2.4.1 Nombres

Ya hemos visto que se usan nombres para denominar a los programas (`ametrico`) y para denominar a las funciones dentro de un módulo (`main`). Cuando queremos dar nombres a valores usamos variables (`millas`, `pies`, `pulgadas`, `metros`). Todos esos nombres se llaman *identificadores* y Python tiene reglas sobre qué es un identificador válido y qué no lo es.

Un identificador comienza con una letra o con guión bajo (_) y luego sigue con una secuencia de letras, números y guiones bajos. Los espacios no están permitidos dentro de los identificadores.

Los siguientes son todos identificadores válidos de Python:

- `hola`
- `hola12t`
- `_hola`
- `Hola`

Python distingue mayúsculas de minúsculas, así que `Hola` es un identificador y `hola` es otro identificador.

Por convención, no usaremos identificadores que empiezan con mayúscula.

Los siguientes son todos identificadores inválidos de Python:

- `hola a12t`
- `8hola`
- `hola\%`
- `Hola*9`

Python reserva 31 palabras para describir la estructura del programa, y no permite que se usen como identificadores. Cuando en un programa nos encontramos con que un nombre no es admitido pese a que su formato es válido, seguramente se trata de una de las palabras de

esta lista, a la que llamaremos de *palabras reservadas*. Esta es la lista completa de las palabras reservadas de Python:

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

2.4.2 Expresiones

Una *expresión* es una porción de código Python que produce o calcula un *valor* (resultado).

- La expresión más sencilla es un valor *literal*. Por ejemplo, la expresión `12345` produce el valor numérico 12345.
- Una expresión puede ser una *variable*, y el valor que produce es el que tiene asociada la variable en el estado. Por ejemplo, si `x` → 5 en el estado, entonces el resultado de la expresión `x` es el valor 5.
- Las expresiones que trabajan con números podemos decir que son expresiones numéricas. Podemos combinar expresiones de este tipo para realizar distintas operaciones y ello lo hacemos usando los operandos numéricos (provistos por Python en este caso) que son, entre otros: +, -, *, /, //, **, %.
- Usamos *operaciones* para combinar expresiones y construir expresiones más complejas:
  - Si `x` es como antes, `x + 1` es una expresión cuyo resultado es 6.
  - Si en el estado `millas` → 1, `pies` → 0 y `pulgadas` → 0, entonces `1609.344 * millas + 0.3048 * pies + 0.0254 * pulgadas` es una expresión cuyo resultado es 1609.344.
  - La exponenciación se representa con el símbolo **. Por ejemplo, `x**3` significa x^3.
  - Se pueden usar paréntesis para indicar un orden de evaluación: `((b * b) - (4 * a * c)) / (2 * a)`.
  - Igual que en la notación matemática, si no hay paréntesis en la expresión, primero se agrupan las exponenciaciones, luego los productos y cocientes, y luego las sumas y restas.
  - Hay que prestar atención con lo que sucede con los cocientes:
    - La expresión `6 / 4` produce el valor `1.5`.
    - La expresión `6 // 4` produce el valor `1`, que es el resultado de la *división entera* entre 6 y 4.
    - La expresión `6 % 4` produce el valor `2`, que es el *resto de la división entera* entre 6 y 4.

  Como vimos en la sección 1.4, los números pueden ser tanto enteros (`0`, `111`, `-24`, almacenados internamente en forma exacta), como reales (`0.0`, `12.5`, `-12.5`, representados internamente en forma aproximada como números *de punto flotante*). Dado que los números enteros y reales se representan de manera diferente, se comportan de manera diferente frente a las operaciones. En Python, los números enteros se denominan `int` (de *integer*), y los números reales `float` (de *floating point*).

- Una expresión puede ser una *llamada a una función*: si `f` es una función que recibe un parámetro, y `x` es una variable, la expresión `f(x)` produce el valor que devuelve la función `f` al invocarla pasándole el valor de `x` por parámetro. Algunos ejemplos son `input()` que produce el valor ingresado por teclado tal como se lo digita y otro ejemplo es `abs(x)`, que produce el valor absoluto del número pasado por parámetro.

2.4.3 No sólo de números viven los programas

No sólo tendremos expresiones numéricas en un programa Python. También puede haber expresiones que sean una *cadena de caracteres* (letras, dígitos, símbolos, etc.), por ejemplo `"Ana"`.

Como en la sección anterior, veremos las reglas de qué constituyen expresiones con caracteres:

- Una expresión puede ser simplemente una cadena de texto. El resultado de la expresión literal `'Ana'` es precisamente el valor `'Ana'`.
- Una variable puede estar asociada a una cadena de texto: si `amiga` → `'Ana'` en el estado, entonces el resultado de la expresión `amiga` es el valor `'Ana'`.
- Se puede usar comillas simples o dobles para representar cadenas simples: `'Ana'` y `"Ana"` son equivalentes.
- Se puede usar tres comillas (simples o dobles) para representar cadenas que incluyen más de una línea de texto:

```
martin_fierro = """Aquí me pongo a cantar
al compás de la vigüela,
que al hombre que lo desvela
una pena estraordinaria,
como el ave solitaria
con el cantar se consuela."""
```

- Usamos operaciones para combinar expresiones y construir expresiones más complejas, pero atención con qué operaciones están permitidas sobre cadenas:
  - El signo `+` no representa la suma sino la *concatenación* de cadenas: Si `amiga` es como antes, `amiga + 'Laura'` es una expresión cuyo valor es `AnaLaura`.

> **⚠ Atención**
>
> No se puede sumar cadenas con números.
>
> ```
> >>> amiga="Ana"
> >>> amiga+'Laura'
> 'AnaLaura'
> >>> amiga+3
> Traceback (most recent call last):
>   File "<stdin>", line 1, in <module>
> TypeError: cannot concatenate 'str' and 'int' objects
> >>>
> ```

  - El signo `*` permite repetir una cadena una cantidad de veces: `amiga * 3` es una expresión cuyo valor es `'AnaAnaAna'`.

Atención

No se pueden multiplicar cadenas entre sí

```
>>> amiga * 3
'AnaAnaAna'
>>> amiga * amiga
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type 'str'
```

2.4.4 Instrucciones

Las *instrucciones* son las órdenes que entiende Python. En general cada línea de un programa Python corresponde a una instrucción. Algunos ejemplos de instrucciones que ya hemos utilizado:

- La instrucción de asignación `<nombre> = <valor>`.
- La instrucción `<nombre> = input(<texto>)`, que permite al usuario ingresar un valor por teclado.
- La instrucción `print(<algo>)`, que permite mostrar algo en pantalla.
- La instrucción más simple que hemos utilizado es la que contiene una única `<expresión>`, y el efecto de dicha instrucción es que Python evalúa la expresión y descarta su resultado. El siguiente es un programa válido en el que todas las instrucciones son del tipo `<expresión>`:

```
0
23.9
abs(-10)
"Este programa no hace nada útil :("
```

2.5 Una guía para el diseño

En su artículo “How to program it”, Simon Thompson plantea algunas preguntas a sus alumnos que son muy útiles para la etapa de diseño:

- ¿Has visto este problema antes, aunque sea de manera ligeramente diferente?
- ¿Conoces un problema relacionado? ¿Conoces un programa que pueda ser útil?
- Observa la especificación. Intenta encontrar un problema que te resulte familiar y que tenga la misma especificación o una parecida.
- Supongamos que hay un problema relacionado, y que ya fue resuelto. ¿Puedes usarlo? ¿Puedes usar sus resultados? ¿Puedes usar sus métodos? ¿Puedes agregarle alguna parte auxiliar a ese programa del que ya dispones?
- Si no puedes resolver el problema propuesto, intenta resolver uno relacionado. ¿Puedes imaginarte uno relacionado que sea más fácil de resolver? ¿Uno más general? ¿Uno más específico? ¿Un problema análogo?

- ¿Puedes resolver una parte del problema? ¿Puedes sacar algo útil de los datos de entrada? ¿Puedes pensar qué información es útil para calcular las salidas? ¿De qué manera se puede manipular las entradas y las salidas de modo tal que estén "más cerca" unas de las otras?
- ¿Utilizaste todos los datos de entrada? ¿Utilizaste las condiciones especiales sobre los datos de entrada que aparecen en el enunciado? ¿Has tenido en cuenta todos los requisitos que se enuncian en la especificación?

2.6 Calidad de software

Los programas que hemos construido hasta ahora son pequeños y simples. Existen proyectos de software profesionales de tamaños muy diversos, yendo desde programas sencillos desarrollados por una única persona hasta proyectos gigantescos, con millones de líneas de código y desarrollados durante años por miles de personas.

Sabías que...

Uno de los proyectos de código abierto más colosales es el núcleo del sistema operativo Linux. Fue publicado por primera vez en 1991, y aun hoy sigue en desarrollo activo. El código fuente es público[a], y cualquiera puede contribuir aportando mejoras. Hasta la versión 4.13 publicada en 2017 participaron más de 15 000 personas, creando en total más de 24 millones de líneas de código.

[a] https://github.com/torvalds/linux

Cuanto más grande es un proyecto de software, más difícil es su construcción y mantenimiento, y más tenemos que prestar atención a la *calidad* con la que está construido. Presentamos aquí una lista no completa de propiedades que contribuyen a la calidad, y algunas preguntas que podemos hacer para medir cuánto contribuye cada factor:

- **Confiabilidad:** ¿El sistema resuelve el problema inicial en forma correcta? ¿Lo resuelve siempre o a veces falla? ¿Cuántas veces falla en un período de tiempo?
- **Testabilidad:** ¿Qué tan fácil es probar que el sistema funciona correctamente? ¿Hay algún proceso de pruebas automáticas o manuales?
- **Performance:** ¿Cuánto tarda el sistema en producir un resultado? ¿Cuántos recursos consume (memoria, espacio en disco, etc.)?
- **Usabilidad:** ¿Puede un nuevo usuario aprender a utilizar el sistema fácilmente? ¿Las operaciones más comunes son fáciles de realizar?
- **Mantenibilidad:** ¿Qué tan legible y entendible es el código? ¿Qué tan fácil es modificar el comportamiento del programa o agregar nuevas funcionalidades?
- **Escalabilidad:** ¿Cómo se comporta el sistema cuando se incrementa la demanda (cantidad de usuarios, cantidad de datos, etc.)?
- **Portabilidad:** ¿El sistema puede funcionar en diferentes plataformas (arquitecturas de procesador, sistemas operativos, navegadores web, etc.)?
- **Seguridad:** ¿Los datos sensibles están protegidos de ataques informáticos? ¿Qué tan difícil es para un atacante tomar el control, desestabilizar o dañar el sistema?

Por supuesto, cada proyecto es particular y algunos de las propiedades mencionadas tendrán más o menos prioridad según el caso. En particular en este curso nos concentraremos más en que nuestros programas sean confiables y mantenibles, y también prestaremos atención a la performance (sobre todo al comparar diferentes algoritmos).

Unidad 3

Funciones

3.1 Creación de funciones

En la primera unidad vimos que el programador puede definir nuevas instrucciones, que llamamos *funciones*. Una función es un fragmento de programa que permite efectuar una operación determinada.

Una función puede recibir ninguno, uno o más parámetros (expresados entre paréntesis, y separados por comas), efectúa una operación, y puede o no devolver un resultado.

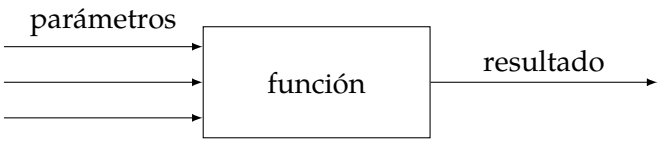


Figura 3.1: Una función recibe parámetros y devuelve un resultado.

Si queremos crear una función (que llamaremos `hola_marta`) que devuelve la cadena de texto "Hola Marta! Estoy programando en Python.", lo que debemos hacer es ingresar el siguiente conjunto de líneas en Python:

```
>>> def hola_marta(): ❶
...     return "Hola Marta! Estoy programando en Python." ❷
...
>>>
```

❶ `def hola_marta():` le indica a Python que estamos escribiendo una función cuyo nombre es `hola_marta`, y los paréntesis indican que la función no recibe ningún parámetro.

❷ La instrucción `return <expresion>` indica cuál será el resultado de la función.

La sangría[1] con la que se escribe la línea `return` es importante: le indica a Python que estamos escribiendo el *cuerpo* de la función (es decir, las instrucciones que la componen), que podría estar formado por más de una sentencia. La línea en blanco que dejamos luego de la instrucción `return` le indica a Python que terminamos de escribir la función (y por eso aparece nuevamente el *prompt*).

Si ahora queremos que la máquina ejecute la función `hola_marta`, debemos escribir `hola_marta()` a continuación del *prompt* de Python:

[1] La sangría puede ingresarse utilizando dos o más espacios, o presionando la tecla Tab. Es importante prestar atención en no mezclar espacios con tabs, para evitar "confundir" al intérprete.

```
>>> hola_marta()
'Hola Marta! Estoy programando en Python.'
>>>
```

Se dice que estamos *invocando* a la función `hola_marta`. Al invocar una función, se ejecutan las instrucciones que habíamos escrito en su cuerpo.

Nuestro amigo Pablo seguramente se pondrá celoso porque escribimos una función que saluda a Marta, y nos pedirá que escribamos una función que lo salude a él. Y así procederemos entonces:

```
>>> def hola_pablo():
...     return "Hola Pablo! Estoy programando en Python."
```

Pero, si para cada amigo que quiere que lo saludemos debemos que escribir una función distinta, parecería que la computadora no es una gran solución. A continuación veremos, sin embargo, que podemos llegar a escribir una única función que se personalice en cada invocación, para saludar a quien queramos. Para eso están precisamente los parámetros.

Escribamos entonces una función `hola` que nos sirva para saludar a cualquiera, de la siguiente manera:

```
>>> def hola(alguien):
...     return "Hola " + alguien + "! Estoy programando en Python."
```

La función `hola` recibe un único *parámetro* (`alguien`). Para llamar a una función debemos asociar cada uno de los parámetros con algún valor determinado (que se denomina *argumento*). Por ejemplo, podemos invocar a la función `hola` dos veces, para saludar a Ana y a Juan, haciendo que `alguien` se asocie al valor `"Ana"` en la primera llamada y al valor `"Juan"` en la segunda. La función en cada caso devolverá un *resultado* que que se calcula a partir del argumento.

```
>>> hola("Ana")
'Hola Ana! Estoy programando en Python.'
>>> hola("Juan")
'Hola Juan! Estoy programando en Python.'
```

Problema 3.1.1. Escribir una función que calcule el cuadrado de un número dado.

Solución.

```
def cuadrado(n):
    return n * n
```

Para invocarla, deberemos hacer:

```
>>> cuadrado(5)
25
```

Problema 3.1.2. Piensa un número, duplícalo, súmale 6, divídelo por 2 y resta el número que elegiste al comienzo. El número que queda es siempre 3.

Solución. Si bien es muy sencillo probar matemáticamente que el resultado de la secuencia de operaciones será siempre 3 sin importar cuál sea el número elegido, podemos aprovechar nuestros conocimientos de programación y probarlo empíricamente.

Para esto escribamos una función que reciba el número elegido y devuelva el número que queda luego de efectuar las operaciones:

```
def f(elegido):
    return ((elegido * 2) + 6) / 2 - elegido
```

Tal vez el cuerpo de la función quedó poco entendible. Podemos mejorarlo dividiendo la secuencia de operaciones en varias sentencias más pequeñas:

```
def f(elegido):
    n = elegido * 2
    n = n + 6
    n = n / 2
    n = n - elegido
    return n
```

Aquí utilizamos una variable llamada `n` y luego en cada sentencia vamos reemplazando el valor de `n` por un valor nuevo.

Las dos soluciones que presentamos son equivalentes. Veamos si al invocar a `f` con distintos números siempre devuelve 3 o no:

```
>>> f(9)
3.0
>>> f(4)
3.0
>>> f(118)
3.0
>>> f(165414606)
3.0
>>> f(0)
3.0
>>> f(-15)
3.0
```

3.2 Documentación de funciones

Cada función escrita por un programador realiza una tarea específica. Cuando la cantidad de funciones disponibles para ser utilizadas es grande, puede ser difícil recordar exactamente qué hace cada función. Es por eso que es extremadamente importante documentar en cada función cuál es la tarea que realiza, cuáles son los parámetros que recibe y qué es lo que devuelve, para que a la hora de utilizarla sea lo pueda hacer correctamente.

Por convención, la documentación de una función se coloca en la primera línea del cuerpo de la misma, como una cadena de caracteres (que, como vimos en la sección 2.4.4, es una instrucción que no tiene ningún efecto). Dado que la documentación suele ocupar más de una línea de texto, se acostumbra encerrarla entre tres pares de comillas.

Así, para la función vista en el ejemplo anterior:

```
def hola(alguien):
    """Devuelve un saludo dirigido a la persona indicada por parámetro."""
    return "Hola " + alguien + "! Estoy programando en Python."
```

Sabías que...

Cuando una función definida está correctamente documentada, es posible acceder a su documentación mediante la función `help` provista por Python. Suponiendo que la función `hola` está definida en el archivo `saludo.py`:

```
>>> import saludo
>>> help(saludo.hola)
Help on function hola in module saludo:

hola(alguien)
    Devuelve un saludo dirigido a la persona indicada por parámetro.
```

De esta forma no es necesario mirar el código de una función para saber lo que hace, simplemente llamando a `help` es posible obtener esta información.

En la sección 3.7 se explica qué hace la instrucción `import`.

3.3 Imprimir versus devolver

Supongamos que tenemos una medida de tiempo expresada en horas, minutos y segundos, y queremos calcular la cantidad total de segundos. Cuando nos disponemos a escribir una función en Python para resolver este problema nos enfrentamos con dos posibilidades:

1. *Devolver* el resultado con la instrucción `return`.
2. *Imprimir* el resultado llamando a la función `print`.

A continuación mostramos ambas implementaciones:

```
def devolver_segundos(horas, minutos, segundos):
    """Transforma en segundos una medida de tiempo expresada en
       horas, minutos y segundos"""
    return 3600 * horas + 60 * minutos + segundos

def imprimir_segundos(horas, minutos, segundos):
    """Imprime una medida de tiempo expresada en horas, minutos y
       segundos, luego de transformarla en segundos"""
    print(3600 * horas + 60 * minutos + segundos)
```

Veamos si funcionan:

```
>>> devolver_segundos(1, 10, 10)
4210
>>> imprimir_segundos(1, 10, 10)
4210
```

Aparentemente el comportamiento de ambas funciones es idéntico, pero hay una gran diferencia. La función `devolver_segundos` nos permite hacer algo como esto:

```
>>> s1 = devolver_segundos(1, 10, 10)
>>> s2 = devolver_segundos(2, 32, 20)
>>> s1 + s2
13350
```

En cambio, la función `imprimir_segundos` nos impide utilizar el resultado de la llamada para hacer otras operaciones; lo único que podemos hacer es mostrarlo en pantalla. Por eso decimos

que `devolver_segundos` es más *reutilizable*. Por ejemplo, podemos reutilizar `devolver_segundos` en la implementación de `imprimir_segundos`, pero no a la inversa:

```
def imprimir_segundos(horas, minutos, segundos):
    """Imprime una medida de tiempo expresada en horas, minutos y
       segundos, luego de transformarla en segundos"""
    print(devolver_segundos(horas, minutos, segundos))
```

Contar con funciones es de gran utilidad, ya que nos permite ir armando una biblioteca de soluciones a problemas simples, que se pueden reutilizar en la resolución de problemas más complejos, tal como lo sugiere Thompson en "How to program it".

En este sentido, más útil que tener una biblioteca donde los resultados se imprimen por pantalla, es contar con una biblioteca donde los resultados se devuelven, para poder manipular los resultados de esas funciones a voluntad: imprimirlos, usarlos para realizar cálculos más complejos, etc.

En general, una función es más reutilizable si devuelve un resultado (utilizando `return`) en lugar de imprimirlo (utilizando `print`). Análogamente, una función es más reutilizable si recibe parámetros en lugar de leer datos mediante la función `input`.

3.4 Cómo usar una función en un programa

Las funciones son útiles porque nos permiten encapsular y repetir una operación (puede que con argumentos distintos) todas las veces que las necesitemos en un programa, sin tener que reescribir la lista de pasos para realizar la operación cada vez.

Supongamos que necesitamos un programa que permita transformar tres duraciones de tiempo en segundos:

1. **Análisis:** El programa debe pedir al usuario tres duraciones expresadas en horas, minutos y segundos, y la tiene que mostrar en pantalla expresada en segundos.
2. **Especificación:**
   - **Entradas:** Tres duraciones leídas de teclado y expresadas en horas, minutos y segundos.
   - **Salidas:** Mostrar por pantalla las duraciones ingresadas, convertida a segundos. Para el juego de datos de entrada (h, m, s) se obtiene entonces $3600h + 60m + s$, y se muestra ese resultado por pantalla.
3. **Diseño:**
   - Se tienen que leer por teclado tres datos y el juego de datos convertirlo a segundos. En pseudocódigo:

```
Leer cuántas horas tiene el tiempo dado
 (y referenciarlo con la variable h)

Leer cuántos minutos tiene tiene el tiempo dado
 (y referenciarlo con la variable m)

Leer cuántos segundos tiene el tiempo dado
 (y referenciarlo con la variable s)
```

```
Mostrar por pantalla 3600 * h + 60 * m + s
```

Pero la conversión a segundos es exactamente lo que hace nuestra función `devolver_segundos`. Si la renombramos a `a_segundos`, podemos hacer que se diseñe como:

```
Leer cuántas horas tiene la duración dada
 (y referenciarlo con la variable h)

Leer cuántos minutos tiene tiene la duración dada
 (y referenciarlo con la variable m)

Leer cuántas segundos tiene la duración dada
 (y referenciarlo con la variable s)

Invocar la función a_segundos(h, m, s) y
mostrar el resultado en pantalla.
```

- El pseudocódigo final queda:

```
Leer cuántas horas tiene la duración dada
 (y referenciarlo con la variable h)

Leer cuántos minutos tiene la duración dada
 (y referenciarlo con la variable m)

Leer cuántos segundos tiene la duración dada
 (y referenciarlo con la variable s)

Invocar la función a_segundos(h, m, s) y
mostrar el resultado en pantalla.
```

4. **Implementación:** A partir del diseño, se escribe el programa Python que se muestra en el Código 3.1, que se guardará en el archivo `tres_tiempos.py`.

Código 3.1 `tres_tiempos.py`**:** Lee tres tiempos y los imprime en segundos

```
1  def a_segundos(horas, minutos, segundos):
2      """Transforma en segundos una medida de tiempo expresada en
3         horas, minutos y segundos"""
4      return 3600 * horas + 60 * minutos + segundos
5
6  def main():
7      """Lee tres tiempos expresados en horas, minutos y segundos,
8         y muestra en pantalla su conversión a segundos"""
9      h = int(input("Cuantas horas?: "))
10     m = int(input("Cuantos minutos?: "))
11     s = int(input("Cuantos segundos?: "))
12     print("Son", a_segundos(h, m, s), "segundos")
13
14 main()
```

Nota. En nuestra implementación decidimos dar el nombre `main` a la función principal del programa. Esto no es más que una convención: "main" significa "principal" en inglés.

5. **Prueba:** Probamos el programa con las ternas $(1,0,0)$, $(0,1,0)$ y $(0,0,1)$:

```
$ python3 tres_tiempos.py
Cuantas horas?: 1
Cuantos minutos?: 0
Cuantos segundos?: 0
Son 3600 segundos
Cuantas horas?: 0
Cuantos minutos?: 1
Cuantos segundos?: 0
Son 60 segundos
Cuantas horas?: 0
Cuantos minutos?: 0
Cuantos segundos?: 1
Son 1 segundos
```

3.5 Alcance de las variables

Ya hemos visto que podemos definir variables, ya sea dentro o fuera del cuerpo de una función. Definamos ahora la siguiente función:

```
>>> def suma_cuadrados(n, m):
...     suma = cuadrado(n) + cuadrado(m)
...     return suma
>>> y = suma_cuadrados(5, 6)
```

¿Qué pasa si intentamos utilizar la variable `suma` fuera de la función?

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

Las variables y los parámetros que se declaran dentro de una función no existen fuera de ella, y por eso se las denomina *variables locales*. Fuera de la función se puede acceder únicamente al valor que devuelve mediante `return`.

Veamos en detalle qué sucede cuando invocamos a la función mediante la instrucción:

```
>>> y = suma_cuadrados(5, 6)
```

1. Se invoca a `suma_cuadrados` con los argumento 5 y 6, y se ejecuta el cuerpo de la función con la variable local `n` → 5 y `n` → 6.
2. La función declara una variable local `suma` → cuadrado(n) + cuadrado(m).
3. Cuando la ejecución llega a la línea `return suma`, la variable `suma` → 61. Por lo tanto, la función devuelve el valor 61.
4. La función termina su ejecución, y con ella dejan de existir todas sus variables locales: `n`, `m` y `suma`.
5. Se declara la variable `y` → 61, que es el valor que devolvió la función.

Si la función no devolviera ningún valor, la variable `y` no quedaría asociada a ningún valor[2].

[2] Técnicamente, quedaría asociada con un valor especial llamado `None`.

3.6 Devolver múltiples resultados

Problema 3.1. Escribir una función que, dada una duración en segundos sin fracciones (representada por un número entero), calcule la misma duración en horas, minutos y segundos.

Solución. La especificación es sencilla:

- La cantidad de horas es la duración informada en segundos dividida por 3600 (división entera).
- La cantidad de minutos es el resto de la división del paso 1, dividido por 60 (división entera).
- La cantidad de segundos es el resto de la división del paso 2.
- Es importante notar que si la duración no se informa como un número entero, todas las operaciones que se indican más arriba carecen de sentido.

¿Cómo hacemos para devolver más de un valor? En realidad lo que se espera de esta función es que devuelva una terna de valores: si ya calculamos `h`, `m` y `s`, lo que debemos devolver es la terna `(h, m, s)`:

```
def a_hms(segundos):
    """Dada una duración entera en segundos
       se la convierte a horas, minutos y segundos"""
    h = segundos // 3600
    m = (segundos % 3600) // 60
    s = (segundos % 3600) % 60
    return h, m, s
```

Esto es lo que sucede al invocar esta función:

```
>>> h, m, s = a_hms(3661)
>>> print("Son", h, "horas", m, "minutos", s, "segundos")
Son 1 horas 1 minutos 1 segundos
```

Sabías que...

Cuando la función debe devolver múltiples resultados, se empaquetan todos juntos en una *n-upla* (secuencia de valores separados por comas) del tamaño adecuado.

Esta característica está presente en Python, Ruby, Haskell y algunos otros pocos lenguajes. En los lenguajes en los que esta característica no está presente, como C, Pascal o Java, es necesario recurrir a otras técnicas más complejas para poder obtener un comportamiento similar.

Respecto de la variable que hará referencia al resultado de la invocación, se podrá usar tanto una n-upla de variables, como en el ejemplo anterior (en cuyo caso podremos nombrar en forma separada cada uno de los resultados), o bien se podrá usar una sola variable (en cuyo caso se considerará que el resultado tiene un solo nombre y la forma de una n-upla):

```
>>> hms = a_hms(3661)
>>> print(hms)
(1, 1, 1)
```

Atención

Si se usa una n-upla de variables para referirse a un resultado, la cantidad de variables tiene que coincidir con la cantidad de valores que se devuelven.

```
>>> x, y = a_hms(3661)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: too many values to unpack
>>> x, y, w, z = a_hms(3661)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: need more than 3 values to unpack
```

3.7 Módulos

A medida que los programas se hacen más grandes y complejos suele ser conveniente dividirlos en *módulos*. Cada uno de los programas que escribimos hasta ahora están formados por un único módulo, ya que cada archivo `.py` es un módulo.

Código 3.2 `saludos.py`**:** Módulo con funciones para saludar

```
def hola(nombre)
    return "Hola, " + nombre

def adios(nombre)
    return "Adiós, " + nombre
```

Código 3.3 `main.py`**:** Módulo principal del programa

```
import saludos

def main()
    nombre = input("¿Cuál es tu nombre?")
    print(saludos.hola(nombre))
    print(saludos.adios(nombre))

main()
```

En Código 3.2 y Código 3.3 se muestra un ejemplo de un programa formado por dos módulos, `saludos` y `main`:

- El módulo `saludos` define dos funciones: `hola` y `adios`. Notar que lo único que hacemos es definir funciones pero nunca las llamamos, justamente porque las vamos a invocar desde el módulo `main`.
- Lo primero que hacemos en el módulo `main` es utilizar la instrucción de Python `import saludos`, para indicar al intérprete que queremos utilizar las funciones definidas en el módulo `saludos`. Luego las invocamos, con la diferencia de que tenemos que anteceder el

nombre de cada función con el nombre del módulo y un ".", en este caso `saludos.hola` y `saludos.chau`. Y finalmente llamamos a la función `main()`.

Para ejecutar el programa lo hacemos con el comando `python main.py`. Cuando el intérprete encuentre la instrucción `import saludo` automáticamente buscará el archivo `saludos.py` y lo ejecutará.

3.7.1 Módulos estándar

Se dice que "Python viene con las baterías incluidas". Esto es porque el intérprete incluye un conjunto numeroso de módulos ya implementados con utilidades de uso general: matemática, acceso al sistema operativo y la red, depuración, criptografía, compresión, interfaces gráficas... ¡Incluso hay una tortuga!

Sabías que...

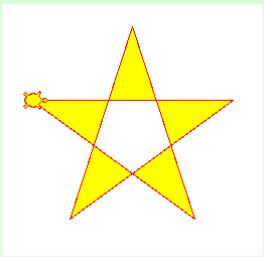
El lenguaje de programación *Logo*, creado en 1967 y utilizado principalmente con fines educativos, introdujo la idea de crear dibujos utilizando la metáfora de una *tortuga* que se mueve por la pantalla obedeciendo a comandos simples.

El módulo `turtle` de Python nos permite crear dibujos usando un sistema muy similar al de Logo:

```
import turtle

def movete_tortu():
    turtle.forward(200)
    turtle.right(144)

turtle.shape("turtle")
turtle.color('red', 'yellow')
turtle.begin_fill()
movete_tortu()
movete_tortu()
movete_tortu()
movete_tortu()
movete_tortu()
turtle.end_fill()
turtle.done()
```



La lista completa de módulos incluidos y sus respectivas instrucciones de uso se puede ver en https://docs.python.org/3/library/index.html.

3.8 Introducción al diseño de pruebas de código[3]

Una de las fases más importantes de la programación, fundamental para el desarrollo de software de calidad, consiste en la etapa de pruebas, la cual mencionamos en el Capítulo 2 como **Probar el programa**.

[3]Esta sección es una adaptación de *El Libro de Python*, disponible en https://ellibrodepython.com/

Esta etapa, en muchas ocasiones, insume más tiempo que escribir el código. En este breve apartado, vamos a introducir algunas formas básicas de testear el código en Python.

Las pruebas, o *test*, se dividen escencialmente en tests manuales y tests automatizados:

- Los tests manuales son tests ejecutados manualmente por una persona, probando diferentes combinaciones y viendo que el comportamiento del código es el esperado.
- Sin embargo, existen muchas utilidades -especialmente importantes en proyectos grandes- para la realización de tests automáticos. Estos consisten en código que testea que otro código se comporta correctamente. La ejecución es automática, y permite ejecutar gran cantidad de verificaciones en muy poco tiempo.

Imaginemos ahora que queremos probar la función `cuadrado(n)` que vimos al inicio de esta Unidad:

Código 3.4 `cuadrado.py`**:** Función para calcular el cuadrado de n

```
def cuadrado(n):
    return n * n
```

Una vez que escribimos nuestro código, lo primero que tenemos que hacer antes de ponerlo en producción (utilizable para los usuarios) es realizar alguna verificación.

La verificación más sencilla que podemos realizar es a través de un *test manual*, probando con un conjunto de datos para verificar que la función hace lo que se espera de ella:

```
>>> print(cuadrado(2))
4
>>> print(cuadrado(6))
36
>>> print(cuadrado(10))
100
```

Como mencionamos antes, los *test manuales* funcionan bien en programas con pocas líneas de código y donde sólo trabajemos nosotros. Sin embargo, a medida que el proyecto crece resultan insuficientes. Para estos casos, es conveniente pensara en *test automáticos*.

Python ofrece muchas herramientas que nos permiten escribir tests que son ejecutados automáticamente, y que si fallan darán un error, alertando al programador de que algo no funciona.

En este curso sólo veremos la más sencilla implementación de test automáticos, la cual funciona a través de la instrucción `assert`, de la siguiente manera:

- Por un lado está la invocación a la función que queremos testear, que devuelve un resultado.
- Por otro lado tenemos el resultado esperado, que comparamos con el resultado devuelto por la función. Si no es igual, se lanza un error.

Podemos automatizar los test anteriores rápidamente de la siguiente forma:

```
>>> assert(cuadrado(2) == 4)
>>> assert(cuadrado(6) == 36)
>>> assert(cuadrado(10) == 100)
```

En este caso, dado que la función `cuadrado()` funciona correctamente, el código se ejecuta si devolver ningún resultado. Si por cualquier motivo alguien modificara nuestra función `cuadrado()` de forma inadecuada, cuando los tests se ejecuten lanzaran una excepción:

```
>>> assert(cuadrado(2) == 4)
Traceback (most recent call last):
    File "ejemplo.py", line 1, in <module>
    assert(cuadrado(2) == 4)
AssertionError
```

De hecho, es posible definir nuestros test dentro de una función específica que se encargará de probar determinadas porciones de código de la siguiente manera:

```
def test_cuadrado():
    print("Testeando cuadrado... ", end="")

    assert(cuadrado(2) == 4)
    assert(cuadrado(6) == 36)
    assert(cuadrado(10) == 100)

    print("Pasó!")
```

De esta forma, cuando ejecutemos nuestra función `test_cuadrado()`, la misma devolverá en pantalla el texto `"Testeando cuadrado... Pasó!"` en caso que la función `cuadrado()` funcione correctamente o devolverá un error en caso que la misma devuelva resultados no esperados.

3.9 Resumen

- Una función puede recibir ninguno, uno o más parámetros. Adicionalmente puede leer datos de la entrada del teclado.
- Una función puede no devolver nada, o devolver uno o más valores. Adicionalmente puede imprimir mensajes para comunicarlos al usuario.
- No es posible acceder a las variables definidas dentro de una función desde el programa principal. Si se quiere utilizar algún valor calculado en la función, será necesario devolverlo.
- Cuando una función realice un cálculo o una operación, es preferible que reciba los datos necesarios mediante los parámetros de la función, y que devuelva el resultado. Las funciones que leen datos del teclado o imprimen mensajes son menos reutilizables.
- Es altamente recomendable documentar cada función que se escribe, para poder saber qué parámetros recibe, qué devuelve y qué hace sin necesidad de leer el código.

Referencia Python

`def funcion(param1, param2, param3):`

Permite definir funciones, que pueden tener ninguno, uno o más parámetros. El cuerpo de la función debe estar un nivel de sangría más adentro que la declaración de la función.

```
def funcion(param1, param2, param3):
    # hacer algo con los parametros
```

Documentación de funciones

Si en la primera línea de la función se ingresa una cadena de caracteres, la misma por convención pasa a ser la documentación de la función, que puede ser accedida mendiante el comando `help(funcion)`.

```
def funcion():
    """Esta es la documentación de la función"""
    # hacer algo
```

`return valor`

Dentro de una función se utiliza la instrucción `return` para indicar el valor que la función debe devolver. Una vez que se ejecuta esta instrucción, se termina la ejecución de la función, sin importar si es la última línea o no. Si la función no contiene esta instrucción, no devuelve nada.

`return valor1, valor2, valor3`

Si se desea devolver más de un valor, se los *empaqueta* en una n-upla de valores. Esta n-upla puede o no ser desempaquetada al invocar la función:

```
def f(valor):
    # operar
    return a1, a2, a3

# desempaquetado:
v1, v2, v3 = f(x)
# empaquetado
v = f(y)
```

`import modulo`

Permite utilizar funciones y valores definidos en el módulo especificado. Las referencias deben ser precedidas por el nombre del módulo y "`.`".

```
>>> import math
>>> math.cos(2 * math.pi)
1.0
```

`import modulo as variable`

Hace lo mismo que `import modulo`, pero nos permite llamar al módulo con una variable nombrada por nosotros.

```
>>> import math as matematica
>>> matematica.cos(2 * matematica.pi)
1.0
```

`from modulo import ref1, ref2, ...`

Similar a `import modulo`, pero importando únicamente las funciones y valores especificados, y además eliminando la necesidad de anteponer el nombre del módulo al utilizarlos:

```
>>> from math import cos, pi
>>> cos(2 * pi)
1.0
```

Unidad 4

Decisiones

Problema 4.1. Debemos leer un número y, si el número es positivo, debemos escribir en pantalla el cartel "Número positivo".

Solución. Especificamos nuestra solución: se deberá leer un número x. Si $x > 0$ se escribe el mensaje `"Número positivo"`.

Diseñamos nuestra solución:

1. Solicitar al usuario un número, guardarlo en x.
2. Si $x > 0$, imprimir `"Número positivo"`

Es claro que la primera línea se puede traducir como

```
    x = int(input("Ingrese un número: "))
```

Sin embargo, con las instrucciones que vimos hasta ahora no podemos tomar el tipo de decisiones que nos planteamos en la segunda línea de este diseño.

Para resolver este problema introducimos una nueva instrucción que llamaremos *condicional* y tiene la siguiente forma:

```
if <expresión>:
    <cuerpo>
```

donde `if` es una palabra reservada, la `<expresión>` es una *condición* y el `<cuerpo>` se ejecuta solo si la condición se cumple.

Antes de seguir adelante explicando la instrucción `if`, debemos introducir un nuevo tipo de dato que nos indicará si se da una cierta situación o no. Hasta ahora las expresiones con las que trabajamos fueron de tipo numérica y de tipo texto; pero ahora la respuesta que buscamos es de tipo *sí* o *no*.

4.1 Expresiones booleanas

Además de los tipos numéricos (`int`, `float`), y las cadenas de texto (`str`), Python introduce un tipo de dato llamado *booleano* (`bool`). Una *expresión booleana* o *expresión lógica* puede tomar dos valores posibles: `True` (sí) o `False` (no).

```
>>> n = 3    # n es de tipo 'int' y toma el valor 3
>>> b = True # b es de tipo 'bool' y toma el valor True
```

4.1.1 Expresiones de comparación

En el ejemplo que queremos resolver, la condición que queremos ver si se cumple o no es que `x` sea mayor que cero. Python provee las llamadas *expresiones de comparación* que sirven para comparar valores entre sí, y que por lo tanto permiten codificar ese tipo de pregunta. En particular la pregunta de si `x` es mayor que cero, se codifica en Python como `x > 0`.

De esta forma, `5 > 3` es una expresión booleana (o condición) cuyo valor es `True`, y `5 < 3` también es una expresión booleana, pero su valor es `False`.

```
>>> 5 > 3
True
>>> 3 > 5
False
```

Las expresiones booleanas de comparación que provee Python son las siguientes:

Expresión	Significado
`a == b`	`a` es igual a `b`
`a != b`	`a` es distinto de `b`
`a < b`	`a` es menor que `b`
`a <= b`	`a` es menor o igual que `b`
`a > b`	`a` es mayor que `b`
`a >= b`	`a` es mayor o igual que `b`

Estos operadores son generalmente denominados *operadores relacionales* en la literatura dado que permiten establecer relaciones entre dos valores. A continuación, algunos ejemplos de uso de estos operadores:

```
>>> 6 == 6
True
>>> 6 != 6
False
>>> 6 > 6
False
>>> 6 >= 6
True
>>> 6 > 4
True
>>> 6 < 4
False
>>> 6 <= 4
False
>>> 4 < 6
True
```

4.1.2 Operadores lógicos

De la misma manera que se puede operar entre números mediante las operaciones de suma, resta, etc., también existen tres operadores lógicos para combinar expresiones booleanas: `and` (y), `or` (o) y `not` (no).

El significado de estos operadores es igual al del castellano, pero vale la pena recordarlo:

Expresión	Significado
`a and b`	El resultado es `True` solamente si `a` es `True` y `b` es `True` de lo contrario el resultado es `False`
`a or b`	El resultado es `True` si `a` es `True` o `b` es `True` (o ambos) de lo contrario el resultado es `False`
`not a`	El resultado es `True` si `a` es `False` de lo contrario el resultado es `False`

Algunos ejemplos:

- `a > b and a > c` es verdadero si `a` es simultáneamente mayor que `b` y que `c`.

```
>>> 5 > 2 and 5 > 3
True
>>> 5 > 2 and 5 > 6
False
```

- `a > b or a > c` es verdadero si `a` es mayor que `b` o `a` es mayor que `c`.

```
>>> 5 > 2 or 5 > 3
True
>>> 5 > 2 or 5 > 6
True
>>> 5 > 8 or 5 > 6
False
```

- `not a > b` es verdadero si `a > b` es falso (o sea si `a <= b` es verdadero).

```
>>> 5 > 8
False
>>> not 5 > 8
True
>>> 5 > 2
True
>>> not 5 > 2
False
```

4.2 Comparaciones simples (estructura condicional simple)

Volvemos al problema que nos plantearon: Debemos leer un número y, si el número es positivo, debemos escribir en pantalla el mensaje `"Número positivo"`.

Recordemos la instrucción `if` que acabamos de introducir y que sirve para tomar decisiones simples. Dijimos que su formato general es:

```
if <expresión>:
    <cuerpo>
```

cuyo efecto es el siguiente:

1. Se evalúa la `<expresión>` (que debe ser una expresión lógica).
2. Si el resultado de la expresión es `True` (verdadero), se ejecuta el `<cuerpo>`.

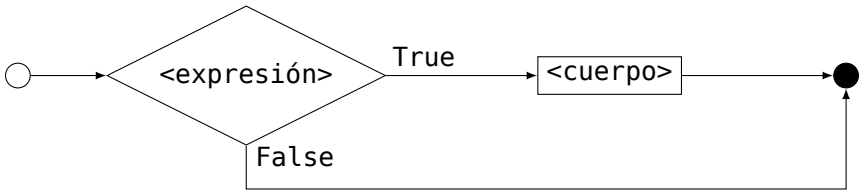


Figura 4.1: Diagrama de flujo para la instrucción `if`.

Esto se puede representar en un *diagrama de flujo*, como el de la Figura 4.1.

Como ahora ya sabemos también cómo construir condiciones de comparación, estamos en condiciones de implementar nuestra solución. Escribimos la función `positivo()` que hace lo pedido:

```
def positivo():
    x = int(input("Ingrese un número: "))
    if x > 0:
        print("Número positivo")
```

y la probamos:

```
>>> positivo()
Ingrese un número: 4
Número positivo
>>> positivo()
Ingrese un número: -25
>>> positivo()
Ingrese un número: 0
```

4.3 Estructura condicional completa

Problema 4.2. Necesitamos además un mensaje `"Número no positivo"` cuando no se cumple la condición.

Modificamos la especificación consistentemente y modificamos el diseño:

1. Solicitar al usuario un número, guardarlo en x.
2. Si $x > 0$, imprimir `"Número positivo"`
3. En caso contrario, imprimir `"Número no positivo"`

La negación de $x > 0$ es $\neg(x > 0)$ que se traduce en Python como `not x > 0`, por lo que implementamos nuestra solución en Python como:

```
def positivo_o_no():
    x = int(input("Ingrese un número: "))
    if x > 0: ❶
        print("Número positivo")
    if not x > 0: ❷
        print("Número no positivo")
```

Probamos la nueva solución y obtenemos el resultado buscado:

```
>>> positivo_o_no()
Ingrese un número: 4
```

```
Número positivo
>>> positivo_o_no()
Ingrese un número: -25
Número no positivo
>>> positivo_o_no()
Ingrese un número: 0
Número no positivo
```

Sin embargo hay algo que nos preocupa: si ya averiguamos una vez, en ❶, si `x > 0`, ¿Es realmente necesario volver a preguntarlo en ❷?.

Existe una construcción alternativa para la estructura de decisión, que tiene la forma:

```
if <expresión>:
    <cuerpo1>
else:
    <cuerpo2>
```

donde `if` y `else` son palabras reservadas. Su efecto es el siguiente:

1. Se evalúa la `<expresión>`.
2. Si el resultado es `True`, se ejecuta el `<cuerpo1>`. En caso contrario, se ejecuta el `<cuerpo2>`.

Volvemos a nuestro diseño:

1. Solicitar al usuario un número, guardarlo en x.
2. Si $x > 0$, imprimir `"Número positivo"`
3. En caso contrario, imprimir `"Número no positivo"`

En la Figura 4.2 se muestra el diagrama de flujo para la estructura `if-else`.

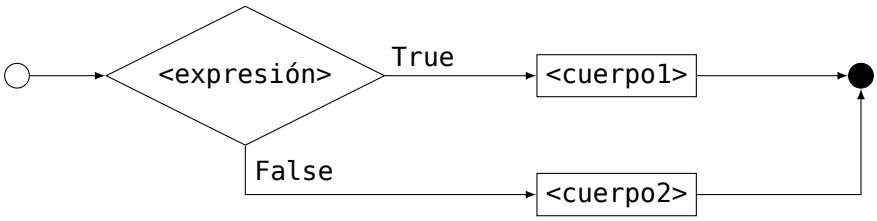


Figura 4.2: Diagrama de flujo para la estructura `if-else`.

Este diseño se implementa como:

```
def positivo_o_no():
    x = int(input("Ingrese un número: "))
    if x > 0:
       print("Número positivo")
    else:
       print("Número no positivo")
```

y lo probamos:

```
>>> positivo_o_no()
Ingrese un número: 4
Número positivo
>>> positivo_o_no()
Ingrese un número: -25
```

```
Número no positivo
>>> positivo_o_no()
Ingrese un número: 0
Número no positivo
```

Es importante destacar que, en general, negar la condición del `if` y poner `else` no son intercambiables, porque no necesariamente producen el mismo efecto en el programa. Notar qué sucede en los dos programas que se transcriben a continuación. ¿Por qué se dan estos resultados?:

```
>>> def pn1():
...     x = int(input("Ingrese un nro: "))
...     if x > 0:
...         print("Número positivo")
...         x = -x
...     if x < 0:
...         print("Número no positivo")
...
>>> pn1()
Ingrese un nro: 25
Número positivo
Número no positivo
```

```
>>> def pn2():
...     x = int(input("Ingrese un nro: "))
...     if x > 0:
...         print("Número positivo")
...         x = -x
...     else:
...         print("Número no positivo")
...
>>> pn2()
Ingrese un nro: 25
Número positivo
```

4.4 Múltiples decisiones anidadas

La decisión de incluir una decisión en un programa, parte de una lectura cuidadosa de la especificación. En nuestro caso la especificación nos decía:

> Si el número es positivo escribir un mensaje `"Número positivo"`, de lo contrario escribir un mensaje `"Número no positivo"`.

Veamos qué se puede hacer cuando se presentan tres o más alternativas:

Problema 4.3. Si el número es positivo escribir un mensaje `"Número positivo"`, si el número es igual a 0 un mensaje `"Igual a 0"`, y si el número es negativo escribir un mensaje `"Número negativo"`.

Una posibilidad es considerar que se trata de una estructura con dos casos como antes, sólo que el segundo caso es complejo (es nuevamente una alternativa):

1. Solicitar al usuario un número, guardarlo en x.
2. Si $x > 0$, imprimir `"Número positivo"`
3. De lo contrario:
   (a) Si $x = 0$, imprimir `"Igual a 0"`
   (b) De lo contrario, imprimir `"Número no positivo"`

Este diseño se implementa como:

```
def pos_cero_o_neg():
    x = int(input("Ingrese un número: "))
    if x > 0:
        print("Número positivo")
```

```
    else:
        if x == 0:
            print("Igual a 0")
        else:
            print("Número negativo")
```

Esta estructura se conoce como de *alternativas anidadas* ya que dentro de una de las ramas de la alternativa (en este caso la rama del `else`) se anida otra alternativa.

Pero ésta no es la única forma de implementarlo. Existe otra construcción, equivalente a la anterior pero que no exige sangrías cada vez mayores en el texto. Se trata de la estructura de *alternativas encadenadas*, que tiene la forma

```
if <expresión_1>:
    <cuerpo_1>
elif <expresión_2>:
    <cuerpo_2>
...
...
elif <expresión_n>:
    <cuerpo_n>
else:
    <cuerpo_else>
```

donde `if`, `elif` y `else` son palabras reservadas.

En nuestro ejemplo:

```
def pos_cero_o_neg():
    x = int(input("Ingrese un número: "))
    if x > 0:
       print("Número positivo")
    elif x == 0:
       print("Igual a 0")
    else:
       print("Número negativo")
```

El efecto de la estructura `if-elif-else` en este ejemplo se muestra en la Figura 4.3.

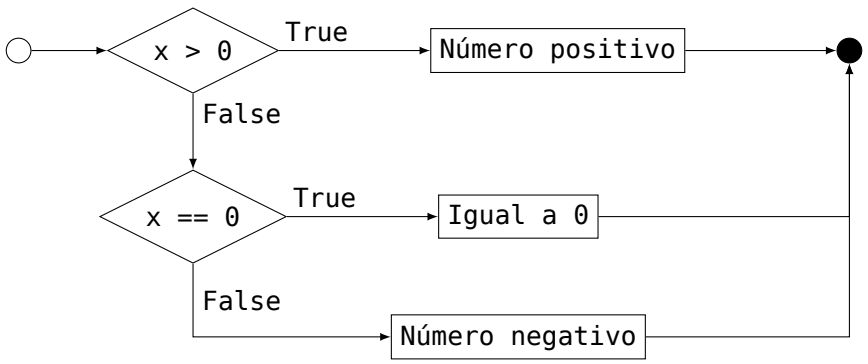


Figura 4.3: Diagrama de flujo para una estructura `if-elif-else`.

Sabías que...

No sólo mediante los operadores vistos (como > o ==) es posible obtener expresiones booleanas. En Python, se consideran *verdaderos* los valores numéricos distintos de 0, las cadenas de caracteres que no son vacías, y en general cualquier valor que no sea 0 o vacío. Los valores nulos o vacíos se consideran *falsos*.

Así, en el ejemplo anterior la línea

```
elif x == 0:
```

también podría escribirse de la siguiente manera:

```
elif not x:
```

Además, en Python existe un valor especial llamado `None` que se utiliza comúnmente para representar la ausencia de un valor. Podemos preguntar si una variable `v` es `None` simplemente con:

```
if v is None:
```

O, como `None` también es considerado un valor nulo,

```
if not v:
```

4.5 Resumen

- Para poder tomar decisiones en los programas y ejecutar una acción u otra, es necesario contar con una **estructura condicional**.
- Las **condiciones** son expresiones *booleanas*, es decir, cuyos valores pueden ser *verdadero* o *falso*, y se las confecciona mediante operadores entre distintos valores.
- Mediante **expresiones lógicas** es posible modificar o combinar expresiones booleanas.
- La estructura condicional puede contar, opcionalmente, con un bloque de código que se ejecuta si no se cumplió la condición.
- Es posible *anidar* estructuras condicionales, colocando una dentro de otra.
- También es posible *encadenar* las condiciones, es decir, colocar una lista de posibles condiciones, de las cuales se ejecuta la primera que sea verdadera.

Referencia Python

`if <condición>:`

Bloque condicional. Las acciones a ejecutar si la condición es verdadera deben tener un mayor nivel de sangría.

```
if <condición>:
    # acciones a ejecutar si condición es verdadera
```

`else:`

Un bloque que se ejecuta cuando no se cumple la condición correspondiente al `if`. Sólo se puede utilizar `else` si hay un `if` correspondiente. Debe escribirse al mismo nivel que `if`, y las acciones a ejecutar deben tener un nivel de sangría mayor.

```
if <condición>:
    # acciones a ejecutar si condición es verdadera
else:
    # acciones a ejecutar si condición es falsa
```

`elif <condición>:`

Bloque que se ejecuta si no se cumplieron las condiciones anteriores pero sí se cumple la condición especificada. Sólo se puede utilizar `elif` si hay un `if` correspondiente, se lo debe escribir al mismo nivel que `if`, y las acciones a ejecutar deben escribirse en un bloque de sangría mayor. Puede haber tantos `elif` como se quiera, todos al mismo nivel.

```
if <condición1>:
    # acciones a ejecutar si condición1 es verdadera
elif <condición2>:
    # acciones a ejecutar si condición2 es verdadera
else:
    # acciones a ejecutar si ninguna condición fue verdadera
```

Operadores de comparación o relacionales

Son los que forman las expresiones booleanas.

Expresión	Significado
`a == b`	`a` es igual a `b`
`a != b`	`a` es distinto de `b`
`a < b`	`a` es menor que `b`
`a <= b`	`a` es menor o igual que `b`
`a > b`	`a` es mayor que `b`
`a >= b`	`a` es mayor o igual que `b`

Operadores lógicos o booleanos

Son los utilizados para concatenar o negar distintas expresiones booleanas.

Expresión	Significado
`a and b`	El resultado es `True` solamente si `a` es `True` y `b` es `True` de lo contrario el resultado es `False`
`a or b`	El resultado es `True` si `a` es `True` o `b` es `True` (o ambos) de lo contrario el resultado es `False`
`not a`	El resultado es `True` si `a` es `False` de lo contrario el resultado es `False`

Unidad 5

Ciclos

5.1 El ciclo definido

Problema 5.1.1. Supongamos que queremos calcular la suma de los primeros 5 números cuadrados.

Solución. Dado que ya tenemos la función `cuadrado` de la Unidad 3, podemos aprovecharla y hacer algo como esto:

```
>>> def suma_5_cuadrados():
    suma = 0
    suma = suma + cuadrado(1)
    suma = suma + cuadrado(2)
    suma = suma + cuadrado(3)
    suma = suma + cuadrado(4)
    suma = suma + cuadrado(5)
    return suma

>>> suma_5_cuadrados()
55
```

Esto resuelve el problema, pero resulta poco satisfactorio. ¿Y si quisiéramos encontrar la suma de los primeros 100 números cuadrados? En ese caso tendríamos que repetir la línea `suma = suma + cuadrado(...)` 100 veces. ¿Se puede hacer algo mejor que esto?

Para resolver este tipo de problema (repetir un cálculo para los valores contenidos en un intervalo dado) de una manera más eficiente, introducimos el concepto de *ciclo definido*. Un ciclo definido es de la forma

```
for <nombre> in <expresión>:
    <cuerpo>
```

El ciclo `for` es una instrucción compuesta ya que incluye una línea de inicialización y un `<cuerpo>`, que a su vez está formado por una o más instrucciones.

Decimos que el ciclo es definido porque una vez evaluada la `<expresión>` (cuyo resultado debe ser una *secuencia de valores*), se sabe exactamente cuántas veces se ejecutará el `<cuerpo>` y qué valores tomará la variable `<nombre>`.

Para resolver el problema de sumar los cuadrados consecutivos en un intervalo necesitamos un ciclo definido que tiene la siguiente forma:

```
for x in range(n1, n2):
    <hacer algo con x>
```

Esta instrucción se lee como:

- Generar la secuencia de valores enteros del intervalo $[n1, n2)$, y
- Para cada uno de los valores enteros que toma `x` en el intervalo generado, se debe hacer lo indicado por `<hacer algo con x>`.

La instrucción que describe el rango en el que va a realizar el ciclo (`for x in range(...)`) es el *encabezado del ciclo*, y las instrucciones que describen la acción que se repite componen el *cuerpo del ciclo*. Todas las instrucciones que describen el cuerpo del ciclo deben tener una sangría mayor que el encabezado del ciclo.

En nuestro ejemplo la secuencia de valores resultante de la expresión `range(n1, n2)` es el intervalo de enteros `[n1, n1+1, ..., n2-1]` y la variable es `x`.

La secuencia de valores se puede indicar como:

- `range(n)`. Establece como secuencia de valores a `[0, 1, ..., n-1]`.
- `range(n1, n2)`. Establece como secuencia de valores a `[n1, n1+1, ..., n2-1]`.
- Se puede definir a mano una secuencia entre corchetes. Por ejemplo,

  ```
  for x in [1, 3, 9, 27]:
      print(x * x)
  ```

  imprimirá los cuadrados de los números 1, 3, 9 y 27.

Solución. Usemos un ciclo definido para resolver el problema anterior de manera más compacta:

```
>>> def suma_5_cuadrados():
...     suma = 0
...     for x in range(1, 6): ❶
...         suma = suma + cuadrado(x)
...     return suma
```

❶ Notar que en nuestro ejemplo necesitamos recorrer todos los valores enteros entre 1 y 5, y el rango generado por `range(n1, n2)` es *abierto* en `n2`. Es decir, `x` tomará los valores `n1`, `n1 + 1`, `n1 + 2`, ..., `n2 - 1`. Por eso es que usamos `range(1, 6)`.

Problema 5.1.2. Hacer una función más genérica que reciba un parámetro `n` y calcule la suma de los primeros `n` números cuadrados.

Solución.

```
>>> def suma_cuadrados(n):
...     suma = 0
...     for x in range(1, n + 1):
...         suma = suma + cuadrado(x)
...     return suma

>>> suma_cuadrados(5)
55
>>> suma_cuadrados(100)
338350
```

Supongamos ahora el siguiente problema:

> Leer un número. Si el número es positivo escribir un mensaje "Numero positivo", si el número es igual a 0 un mensaje "Igual a 0", y si el número es negativo escribir un mensaje "Numero negativo". El usuario debe poder ingresar muchos números y cada vez que se ingresa uno debemos informar si es positivo, cero o negativo.

Utilizando los ciclos definidos vistos en las primeras unidades, es posible preguntarle al usuario cada vez, al inicio del programa, cuántos números va a ingresar para consultar. La solución propuesta resulta:

```
def muchos_pcn():
    i = int(input("Cuantos numeros quiere procesar?: "))
    for j in range(0, i):
        x = int(input("Ingrese un numero: "))
        if x > 0:
            print("Numero positivo")
        elif x == 0:
            print("Igual a 0")
        else:
            print("Numero negativo")
```

Su ejecución es exitosa:

```
>>> muchos_pcn()
Cuantos numeros quiere procesar: 3
Ingrese un numero: 25
Numero positivo
Ingrese un numero: 0
Igual a 0
Ingrese un numero: -5
Numero negativo
>>>
```

Sin embargo, el uso de este programa no resulta muy intuitivo, porque obliga al usuario a contar de antemano cuántos números va a querer procesar, sin equivocarse, en lugar de ingresar uno a uno los números hasta procesarlos a todos.

5.2 Ciclos indefinidos

Para poder resolver este problema sin averiguar primero la cantidad de números a procesar, debemos introducir una instrucción que nos permita construir ciclos que no requieran que se informe de antemano la cantidad de veces que se repetirá el cálculo del cuerpo. Se trata de los *ciclos indefinidos*, en los cuales se repite el cálculo del cuerpo mientras una cierta condición es verdadera.

Un ciclo indefinido es de la forma

```
while <expresión>:
    <cuerpo>
```

donde `while` es una palabra reservada, y la `<expresión>` debe ser booleana, igual que en las instrucciones `if`. El `<cuerpo>` es, como siempre, una o más instrucciones de Python.

El funcionamiento de esta instrucción es el siguiente:

1. Evaluar la condición.
2. Si la condición es falsa, salir del ciclo.
3. Si la condición es verdadera, ejecutar el cuerpo.
4. Volver a 1.

En la Figura 5.1 se muestra el diagrama de flujo correspondiente al ciclo indefinido `while`.

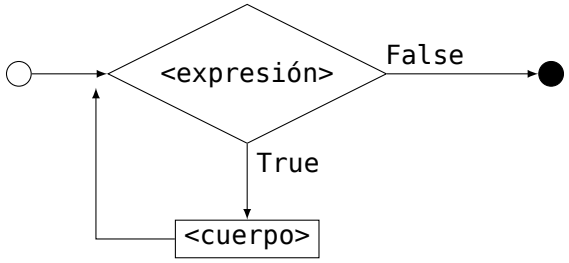


Figura 5.1: Diagrama de flujo para el ciclo indefinido `while`.

5.3 Ciclo interactivo

¿Cuál es la condición y cuál es el cuerpo del ciclo en nuestro problema? Claramente, el cuerpo del ciclo es el ingreso de datos y la verificación de si es positivo, negativo o cero. En cuanto a la condición, es que haya más datos para seguir calculando.

Definimos una variable `hay_mas_datos`, que valdrá "Si" mientras haya datos.

Se le debe preguntar al usuario, después de cada cálculo, si hay o no más datos. Cuando el usuario deje de responder "Si", dejaremos de ejecutar el cuerpo del ciclo.

Una primera aproximación al código necesario para resolver este problema podría ser:

```
def muchos_pcn():
    while hay_mas_datos == "Si":
        x = int(input("Ingrese un numero: "))
        if x > 0:
            print("Numero positivo")
        elif x == 0:
            print("Igual a 0")
        else:
            print("Numero negativo")

        hay_mas_datos = input("¿Quiere seguir? <Si-No>: ")
```

Veamos qué pasa si ejecutamos la función tal como fue presentada:

```
>>> muchos_pcn()
Traceback (most recent call last):
  File "<pyshell#25>", line 1, in <module>
    muchos_pcn()
  File "<pyshell#24>", line 2, in muchos_pcn
    while hay_mas_datos == "Si":
UnboundLocalError: local variable 'hay_mas_datos' referenced before assignment
```

El problema que se presentó en este caso, es que `hay_mas_datos` no tiene un valor asignado en el momento de evaluar la condición del ciclo por primera vez.

Es importante prestar atención a cuáles son las variables que hay que inicializar antes de ejecutar un ciclo, para asegurar que la expresión booleana que lo controla sea evaluable.

Una posibilidad es preguntarle al usario, antes de evaluar la condición, si tiene datos; otra posibilidad es suponer que si llamó a este programa es porque tenía algún dato para calcular, y darle el valor inicial "Si" a `hay_mas_datos`.

Encararemos la segunda opción:

```
def muchos_pcn():
    hay_mas_datos = "Si"
    while hay_mas_datos == "Si":
        x = int(input("Ingrese un numero: "))
        if x > 0:
            print("Numero positivo")
        elif x == 0:
            print("Igual a 0")
        else:
            print("Numero negativo")

        hay_mas_datos = input("Quiere seguir? <Si-No>: ")
```

El esquema del ciclo interactivo es el siguiente:

```
hay_mas_datos hace referencia a "Si"
Mientras hay_mas_datos haga referencia a "Si":
    Pedir datos
    Realizar cálculos
    Preguntar al usuario si hay más datos ("Si" cuando los hay)
    hay_mas_datos hace referencia al valor ingresado
```

Ésta es una ejecución:

```
>>> muchos_pcn()
Ingrese un numero: 25
Numero positivo
Quiere seguir? <Si-No>: Si
Ingrese un numero: 0
Igual a 0
Quiere seguir? <Si-No>: Si
Ingrese un numero: -5
Numero negativo
Quiere seguir? <Si-No>: No
```

5.4 Ciclo con centinela

Un problema que tiene nuestra primera solución es que resulta poco amigable preguntarle al usuario después de cada cálculo si desea continuar. Para evitar esto, se puede usar el método del *centinela*: un valor arbitrario que, si se lee, le indica al programa que el usuario desea salir del ciclo. En este caso, podemos suponer que si el usuario ingresa el caracter *, es una indicación de que desea terminar.

El esquema del ciclo con centinela es el siguiente:

```
Pedir datos
Mientras el dato pedido no coincida con el centinela:
```

```
    Realizar cálculos
    Pedir datos
```

El programa resultante es el siguiente:

```
def muchos_pcn():
    centinela = input("Ingrese un numero (* para terminar): ") ❶

    while centinela != "*":
        x = int(centinela)
        if x > 0:
            print("Numero positivo")
        elif x == 0:
            print("Igual a 0")
        else:
            print("Numero negativo")

        centinela = input("Ingrese un numero (* para terminar): ") ❷
```

Notar que no podemos hacer `centinela = int(input(...))` porque cuando el usuario ingrese `'*'` la llamada a `int` fallaría (al no poder convertir `'*'` a un valor entero). Por eso es que por un lado hacemos la llamada a `input`, y una vez que sabemos que el valor `centinela` no es un `'*'`, lo convertimos a entero llamando a `int`.

Y ahora lo ejecutamos:

```
>>> muchos_pcn()
Ingrese un numero (* para terminar): 25
Numero positivo
Ingrese un numero (* para terminar): 0
Igual a 0
Ingrese un numero (* para terminar): -5
Numero negativo
Ingrese un numero (* para terminar): *
```

El ciclo con centinela es muy claro pero tiene un problema: hay una línea de código repetida, marcada con ❶ y ❷.

Si en la etapa de mantenimiento tuviéramos que realizar un cambio en el ingreso del dato (por ejemplo, cambiar el mensaje) deberíamos estar atentos y corregir ambas líneas. En principio no parece ser un problema muy grave, pero a medida que el programa y el código se hacen más complejos, se hace mucho más difícil llevar la cuenta de todas las líneas de código duplicadas, y por lo tanto se hace mucho más fácil cometer el error de cambiar una de las líneas y olvidar hacer el cambio en la línea duplidada.

> El código duplicado suele incrementar el esfuerzo necesario para hacer modificaciones en la etapa de mantenimiento. Es conveniente prestar atención en a etapa de implementación, y modificar el código para eliminar la duplicación.

Veamos cómo eliminar el código duplicado en nuestro ejemplo. Lo ideal sería leer el dato `centinela` en un único punto del programa. Una opción es *extraer* el código duplicado en una función:

```
def leer_centinela():
    return input("Ingrese un numero (* para terminar): ")

def muchos_pcn():
```

```
    centinela = leer_centinela()
    while centinela != "*":
        x = int(centinela)
        if x > 0:
            print("Numero positivo")
        elif x == 0:
            print("Igual a 0")
        else:
            print("Numero negativo")

        centinela = leer_centinela()
```

Sabías que...

Desde hace mucho tiempo los ciclos infinitos vienen provocando dolores de cabeza a los programadores. Cuando un programa deja de responder y se utiliza todos los recursos de la computadora, suele deberse a que entró en un ciclo del que no puede salir.

Estos bucles pueden aparecer por una gran variedad de causas. A continuación algunos ejemplos de ciclos de los que no se puede salir, siempre o para ciertos parámetros. Queda como ejercicio encontrar el error en cada uno.

```
def menor_factor_primo(x):
    """Devuelve el menor factor primo del número x."""
    n = 2
    while n <= x:
        if x % n == 0:
            return n
```

```
def buscar_impar(x):
    """Divide el número recibido por 2 hasta que sea impar."""
    while x % 2 == 0:
        x = x / 2
    return x
```

5.5 Resumen

- Además de los ciclos definidos, en los que se sabe cuáles son los posibles valores que tomará una determinada variable, existen los ciclos indefinidos, que se terminan cuando no se cumple una determinada condición.
- La condición que termina el ciclo puede estar relacionada con una entrada de usuario o depender del procesamiento de los datos.
- Se puede utilizar el método del *centinela* cuando se quiere que un ciclo se repita hasta que el usuario indique que no quiere continuar.

Referencia Python

`for <nombre> in <expresión>:`

Introduce un ciclo definido. Una vez evaluada la <expresión> (cuyo resultado debe ser una secuencia de valores), se sabe exactamente cuántas veces se ejecutará el <cuerpo> y qué valores tomará la variable <nombre>.

```
for <nombre> in <expresión>:
    # el cuerpo de ejecuta una cantidad definida de veces
    <cuerpo>
```

`while <condicion>:`

Introduce un ciclo indefinido, que se termina cuando la condición sea falsa.

```
while <condición>:
    # acciones a ejecutar mientras condición sea verdadera
```

Unidad 6

Validación

6.1 Errores

En un programa podemos encontrarnos con distintos tipos de errores, pero a grandes rasgos podemos decir que todos los errores pertenecen a una de las siguientes categorías.

- Errores de sintaxis: estos errores son seguramente los más simples de resolver, pues son detectados por el intérprete (o por el compilador, según el tipo de lenguaje que estemos utilizando) al procesar el código fuente y generalmente son consecuencia de equivocaciones al escribir el programa. En el caso de Python estos errores son indicados con un mensaje *SyntaxError*. Por ejemplo, si trabajando con Python intentamos definir una función y en lugar de `def` escribimos `dev`.
- Errores semánticos: se dan cuando un programa, a pesar de no generar mensajes de error, no produce el resultado esperado. Esto puede deberse, por ejemplo, a un algoritmo incorrecto o a la omisión de una sentencia.
- Errores de ejecución: estos errores aparecen durante la ejecución del programa y su origen puede ser diverso. En ocasiones pueden producirse por un uso incorrecto del programa por parte del usuario, por ejemplo si el usuario ingresa una cadena cuando se espera un número. En otras ocasiones pueden deberse a errores de programación, por ejemplo si una función intenta realizar una división por cero. Una causa común de errores de ejecución, que generalmente excede al programador y al usuario, son los recursos externos al programa, por ejemplo si el programa intenta leer un archivo y el mismo se encuentra dañado. Los errores de ejecución son llamados comúnmente *excepciones*.

Tanto a los errores de sintaxis como a los semánticos se los puede detectar y corregir durante la construcción del programa ayudados por el intérprete y la ejecución de pruebas. Pero no ocurre esto con los errores de ejecución, ya que no siempre es posible saber cuándo ocurrirán y puede resultar muy complejo (o incluso casi imposible) reproducirlos. Es por ello que el resto de la unidad nos centraremos en cómo preparar nuestros programas para lidiar con este tipo de errores. En particular, trataremos con una técnica llamada *validación*, que sirve para tratar un tipo especial de errores de ejecución relacionado con los errores de entrada de usuario.

6.2 Validaciones

Las validaciones son técnicas que permiten asegurar que los valores con los que se vaya a operar estén dentro de determinado dominio.

Estas técnicas son particularmente importantes al momento de utilizar entradas del usuario (o entradas externas en general) en nuestro código, y también se las utiliza para comprobar precondiciones. Al uso intensivo de estas técnicas se lo suele llamar *programación defensiva*.

Si bien quien invoca una función debe preocuparse de cumplir con las precondiciones de ésta, si las validaciones están hechas correctamente pueden devolver información valiosa para que el invocante pueda actuar en consecuencia.

Hay distintas formas de comprobar el dominio de un dato. Por ejemplo, se puede comprobar el contenido; o que una variable sea de un tipo en particular.

También se debe tener en cuenta qué hará nuestro código cuando una validación falle, ya que queremos darle información al invocante que le sirva para procesar el error. El error producido tiene que ser fácilmente reconocible.

En cualquier caso, lo importante es que el resultado generado por nuestro código cuando funciona correctamente y el resultado generado cuando falla debe ser claramente distinto.

6.2.1 Comprobaciones por tipo

En esta clase de comprobaciones nos interesa el tipo del dato que vamos a tratar de validar. Python nos indica el tipo de una variable usando la función `type`. Por ejemplo, para comprobar que una variable contenga un tipo entero podemos hacer:

```
if type(x) != int:
    print("El valor debe ser del tipo int")
```

De la misma manera, podemos realizar una comprobación semejante para los otros tipos de datos que vimos durante este curso: `float`, `str` o `bool`.

> **Atención**
>
> Hacer comprobaciones sobre los tipos de las variables suele resultar demasiado restrictivo, ya que es muy posible que una porción de código que opere con un tipo en particular funcione correctamente con otros tipos de variables que se comporten de forma similar.
>
> Es por eso que hay que tener mucho cuidado al limitar el uso de una variable por su tipo, y en muchos casos es preferible limitarlas por sus propiedades.

Otra cuestión a tener en cuesta es que para la mayoría de los tipos básicos de Python existe una función que se llama de la misma manera que el tipo que devuelve un elemento de ese tipo, por ejemplo, `int()` devuelve 0, `dict()` devuelve `{}` y así. Además, estas funciones suelen poder recibir un elemento de otro tipo para tratar de convertirlo, por ejemplo, `int(3.0)` devuelve 3, `list("Hola")` devuelve `['H', 'o', 'l', 'a']`. Esta operación se denomina informalmente *casteo de tipos* en programación.

Usando está conversión conseguimos dos cosas: podemos convertir un tipo recibido al que realmente necesitamos, a la vez que tenemos una copia de este, dejando el original intacto, que es importante cuando estamos tratando con tipos mutables.

Por ejemplo, si se quiere contar con una función de división entera que pueda recibir diversos parámetros, podría hacerse de la siguiente manera.

```
def division_entera(x, y):
    """Calcula la división entera después de convertir los parámetros a
    enteros."""
    return int(x) // int(y)
```

De esta manera, la función `division\_entera` puede ser llamada incluso con cadenas que contengan expresiones enteras. Que este comportamiento sea deseable o no, depende siempre de cada caso.

```
>>> division_entera("5", 4.3)
1
>>> division_entera(5, None)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 1, in division_entera
TypeError: int() argument must be a string, a bytes-like object or a number,
not 'NoneType'
```

6.2.2 Entrada del usuario

En el caso particular de una porción de código que trate con entrada del usuario, no se debe asumir que el usuario vaya a ingresar los datos correctamente, ya que los seres humanos tienden a cometer errores al ingresar información.

Por ejemplo, si se desea que un usuario ingrese un número entero, debemos comprobar el tipo de dato que ingresó. Python, por ejemplo, posee la función `isdecimal` que indica si un string es un número natural.

```
def pedir_numero_natural():
    """Solicita un número natural y lo devuelve.
    Si el valor ingresado no es un natural, imprime un mensaje de error y retorna
    ↪  el string "Error".
    """
    x = input("Ingrese un número natural: ")

    if x.isdecimal():
        x = int(x)
    else:
        print("El valor no es un número natural")
        x = "Error"

    return x
```

Esta función devuelve un número entero, o imprime un mensaje de error y devuelve el string ”Error” si el usuario no ingresó un entero positivo.

Sin embargo, esto no es satisfactorio: si el usuario no ingresa la información correctamente, el programa podría no continuar si dicha información fuese necesaria para la resolución de la tarea del programa.

Es por ello que el proceso de validación debe asegurar que una vez realizada la validación el dato sea válido. Podemos hacerlo solicitando que se vuelva a pedir al usuario que ingrese la información.

De esta forma, el proceso de validación tendría el siguiente flujo:

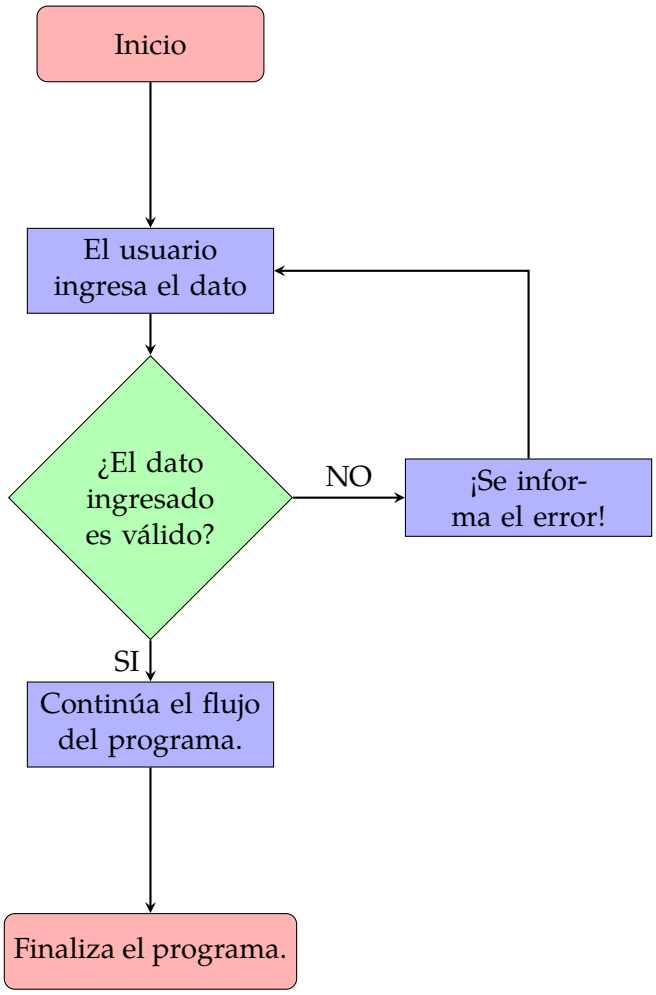


Figura 6.1: Proceso de validación de datos.

```
def pedir_numero_natural():
    """Solicita un número natural y lo devuelve.
    Mientras el valor ingresado no sea un natural, vuelve a solicitarlo.
    """
    x = input("Ingrese un número natural: ")

    while not(x.isdecimal()):
        print(f'{x} no es un número natural')
        x = input("Ingrese un número natural: ")

    return int(x)
```

En determinadas situaciones, podría ser deseable poner un límite a la cantidad máxima de intentos que el usuario tiene para ingresar la información correctamente y, superada esa cantidad máxima de intentos, retornar un valor especial para que sea manejada por el código invocante o imprimir un mensaje de error.

```
def pedir_numero_natural():
    """Solicita un número natural y lo devuelve.
    Si el valor ingresado no es un natural, da 5 intentos para ingresarlo
    ↪ correctamente, y de no ser así, imprime mensaje
```

```
    de error de superación de intentos y devuelve -1."""
    numero_natural = False
    intentos = 0

    while not(numero_natural) and intentos<5:
        x = input("Ingrese un número natural: ")
        if x.isdecimal():
            numero_natural = True
        else:
            intentos+=1
            print(f'{x} no es un número natural')

    if intentos==5:
        print("Valor incorrecto ingresado en 5 intentos")
        x = -1

    return int(x)
```

Por otro lado, cuando la entrada ingresada sea una cadena, no es esperable que el usuario la vaya a ingresar en mayúsculas o minúsculas; ambos casos deben ser considerados.

```
def lee_opcion():
    """Solicita una opción de menú y la devuelve."""
    opcion_invalida = True
    while opcion_invalida:
        opcion = input("Ingrese A (Altas) - B (Bajas) - M (Modificaciones): ")
        opcion = opcion.upper()
        if (opcion == "A" or opcion == "B" or opcion == "M"):
            opcion_invalida = False
        else:
            print(f'La tecla {opcion} no es una opción válida.')

    return opcion
```

6.3 Resumen

- Los errores que se pueden presentar en un programa son: de sintaxis (detectados por el intérprete), de semántica (el programa no funciona correctamente), o de ejecución.
- Antes de actuar sobre un dato en una porción de código, es deseable corroborar que se lo pueda utilizar. Para ello se puede validar su contenido, su tipo o sus atributos.
- Cuando no es posible utilizar un dato dentro de una porción de código, es importante informar el problema al código invocante, lo que puede hacerse mediante un valor de retorno especial.

Unidad 7

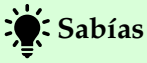
Cadenas de caracteres

Inicialmente, cuando hablamos de tipos de datos simples, hicimos mención al tipo de dato caracter, al cual podemos definir como un único símbolo alfanumérico compuesto por letras, dígitos y símbolos especiales.

En cambio, una cadena de caracteres es una sucesión de caracteres que se almacenan en un área contigua de la memoria, que puede ser leida o escrita.

Ya hemos usado las cadenas para mostrar mensajes, pero sus usos son mucho más amplios que sólo ése: los textos que manipulamos mediante los editores de texto, los textos de Internet que analizan los buscadores y los mensajes enviados mediante correo electrónico son todos ejemplos de cadenas de caracteres.

Para poder programar este tipo de aplicaciones debemos aprender a manipularlas. Comenzaremos a ver ahora cómo hacer cálculos con cadenas.



Sabías que...

En Python todos los valores tienen asignado un *tipo*. La función `type` de Python nos permite averiguar de qué tipo es un valor. Las cadenas son de tipo `str`:

```
>>> type("Hola")
<class 'str'>
```

7.1 Operaciones con cadenas

Ya vimos en la sección 2.4.3 que es posible:

- Sumar cadenas entre sí (y el resultado es la concatenación de todas las cadenas dadas):

```
>>> "Un divertido " + "programa " + "de " + "radio"
'Un divertido programa de radio'
```

- Multiplicar una cadena `s` por un número `k` (y el resultado es la concatenación de `s` consigo misma, `k` veces):

```
>>> 3 * "programas "
'programas programas programas '
```

A continuación, otras operaciones y particularidades de las cadenas.

7.1.1 Obtener la longitud de una cadena

Se puede averiguar la cantidad de caracteres que conforman una cadena utilizando una función provista por Python: `len`.

```
>>> len("programas ")
10
```

Existe una cadena especial, que llamaremos *cadena vacía*, que es la cadena que no contiene ningún carácter (se la indica sólo con un apóstrofe o comilla que abre, y un apóstrofe o comilla que cierra), y que por lo tanto tiene longitud cero:

```
>>> s = ""
>>> s
''
>>> len(s)
0
```

7.1.2 Una operación para recorrer todos los caracteres de una cadena

Python nos permite recorrer todos los caracteres de una cadena de manera muy sencilla, usando directamente un ciclo definido:

```
>>> for c in "programas ":
...     print(c)
...
p
r
o
g
r
a
m
a
s

>>>
```

7.1.3 Preguntar si una cadena contiene una subcadena

El operador `in` nos permite preguntar si una cadena contiene una subcadena. `a in b` es una expresión que se evalúa a `True` si la cadena `b` contiene la subcadena `a`.

```
>>> 'qué' in 'Hola, ¿qué tal?'
True
>>> '7' in '2468'
False
```

Al ser una expresión booleana, podemos utilizarlo como condición de un `if` o un `while`:

```
if "Hola" in s:
    print("Al parecer la cadena s es un saludo")
```

7.1.4 Acceder a una posición de la cadena

Queremos averiguar cuál es el carácter que está en la posición i-ésima de una cadena. Para ello Python nos provee de una notación con corchetes: escribiremos `s[i]` para hablar de la posición i-ésima de la cadena `s`.

Trataremos de averiguar con qué letra empieza una cadena.

```
>>> s = "Veronica"
>>> s[1]
'e'
```

`s[1]` nos muestra la segunda letra, no la primera. ¿Algo falló? No, lo que sucede es que en Python las posiciones se cuentan desde 0.

```
>>> s[0]
'V'
```

Las distintas posiciones de una cadena `s` se llaman *índices*. Los índices son números enteros que pueden tomar valores entre `-len(s)` y `len(s) - 1`.

- Los índices positivos (entre `0` y `len(s) - 1`) son lo que ya vimos: los caracteres de la cadena del primero al útimo.
- Los índices negativos (entre `-len(s)` y `-1`) proveen una notación que hace más fácil indicar cuál es el último carácter de la cadena: `s[-1]` es el último carácter de `s`, `s[-2]` es el penúltimo carácter de `s`, `s[-len(s)]` es el primer carácter de `s`.

Algunos ejemplos de acceso a distintas posiciones en una cadena.

```
>>> s = "Veronica"
>>> len(s)
8
>>> s[0]
'V'
>>> s[7]
'a'
>>> s[8]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>> s[-1]
'a'
>>> s[-8]
'V'
>>> s[-9]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

7.2 Segmentos de cadenas

Python ofrece también una notación para identificar segmentos de una cadena. La notación es similar a la de los rangos que vimos en los ciclos definidos: `s[0:2]` se refiere a la subcadena formada por los caracteres cuyos índices están en el rango `[0,2)`:

```
>>> s[0:2]
'Ve'
>>> s[-4:-2]
'ni'
>>> s[0:8]
'Veronica'
```

Si `j` es un entero no negativo, se puede usar la notación `s[:j]` para representar al segmento `s[0:j]`; también se puede usar la notación `s[j:]` para representar al segmento `s[j:len(s)]`.

```
>>> s[:3]
'Ver'
>>> s[3:]
'onica'
```

Pero hay que tener cuidado con salirse del rango (en particular hay que tener cuidado con la cadena vacía):

```
>>> s[10]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>> s = ""
>>> s
''
>>> len(s)
0
>>> s[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Sin embargo `s[0:0]` no da error. ¿Por qué?

```
>>> s[0:0]
''
```

7.3 Las cadenas son inmutables

Resulta que la persona sobre la que estamos hablando en realidad se llama Veronika, con "k". Como conocemos la notación de corchetes, tratamos de corregir sólo el carácter correspondiente de la variable `s`:

```
>>> s[6] = "k"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

El error que se despliega nos dice que la cadena no soporta la modificación de un carácter. Decimos que *las cadenas son inmutables*.

Si queremos corregir la ortografía de una cadena, debemos hacer que la variable `s` se refiera a una nueva cadena:

```
>>> s = "Veronika"
>>> s
'Veronika'
```

7.4 Procesamiento sencillo de cadenas

Problema 7.1. Nuestro primer problema es muy simple: Queremos contar cuántas letras "A" hay en una cadena `s`.

1. **Especificación:** Dada una cadena `s`, la función retorna un valor `contador` que representa cuántas letras "A" tiene `s`.

2. **Diseño:**

   ¿Se parece a algo que ya conocemos?

   Ante todo es claro que se trata de un ciclo definido, porque lo que hay que tratar es cada uno de los caracteres de la cadena `s`, o sea que estamos frente a un esquema:

   ```
   para cada letra de s
       averiguar si la letra es 'A'
       y tratarla en consecuencia
   ```

   Nos dice la especificación que se necesita una variable `contador` que cuenta la cantidad de letras "A" que contiene `s`. Y por lo tanto sabemos que el tratamiento es: si la letra es "A" se incrementa el contador en 1, y si la letra no es "A" no se lo incrementa, o sea que nos quedamos con un esquema de la forma:

   ```
   para cada letra de s
       averiguar si la letra es 'A'
       y si lo es, incrementar en 1 el contador
   ```

   ¿Estará todo completo? En realidad, en el diseño no planteamos el retorno del valor del contador. Lo completamos entonces:

   ```
   para cada letra de s
       averiguar si la letra es 'A'
       y si lo es, incrementar en 1 el contador
   retornar el valor del contador
   ```

   ¿Y ahora estará todo completo? Ahora vamos a poner manos a la obra y a programar esta solución.

3. **Implementación**

   Ya vimos que Python nos provee de un mecanismo para recorrer una cadena: una instrucción `for` que nos brinda un carácter por vez, del primero al último.

   Proponemos la siguiente solución:

   ```
   def contarA(s):
       for letra in s:
           if letra == "A":
               contador = contador + 1
       return contador
   ```

   Y la probamos

   ```
   >>> contarA("Ana")
   Traceback (most recent call last):
     File "<stdin>", line 1, in <module>
     File "<stdin>", line 4, in contarA
   UnboundLocalError: local variable 'contador' referenced before assignment
   ```

¿Qué es lo que falló? ¡Falló el diseño! Evidentemente la variable `contador` debe tomar un valor inicial antes de empezar a contar las apariciones del caracter "A". Volvamos al diseño entonces.

> Es muy tentador quedarse arreglando la implementación, sin volver al diseño, pero eso es de muy mala práctica, porque el diseño queda mal documentado, y además podemos estar dejando de tener en cuenta otras situaciones erróneas.

4. **Diseño (revisado)** Habíamos llegado a un esquema de la forma

```
para cada letra de s
    averiguar si la letra es 'A'
    y si lo es, incrementar en 1 el contador
retornar el valor del contador
```

¿Cuál es el valor inicial que debe tomar `contador`? Como nos dice la especificación `contador` cuenta la cantidad de letras "A" que tiene la cadena `s`. Pero si nos detenemos en medio de la computación, cuando aún no se recorrió toda la cadena sino sólo los primeros 10 caracteres, por ejemplo, el valor de `contador` refleja la cantidad de "A" que hay en los primeros 10 caracteres de `s`.

Si llamamos *parte izquierda de* `s` al segmento de `s` que ya se recorrió, diremos que cuando leímos los primeros 10 caracteres de `s`, su parte izquierda es el segmento `s[0:10]`.

El valor inicial que debemos darle a `contador` debe reflejar la cantidad de "A" que contiene la parte izquierda de `s` cuando aún no iniciamos el recorrido, es decir cuando esta parte izquierda es `s[0:0]` (o sea la cadena vacía). Pero la cantidad de caracteres iguales a "A" de la cadena vacía es 0.

Por lo tanto el diseño será:

```
inicializar el contador en 0
para cada letra de s
    averiguar si la letra es 'A'
    y si lo es, incrementar en 1 el contador
retornar el valor del contador
```

Lo identificaremos como el esquema *Inicialización - Ciclo de tratamiento - Retorno de valor*. Pasamos ahora a implementar este diseño:

5. **Implementación (del diseño revisado)**

```
1 def contarA(s):
2     """Devuelve cuántas letras "A" aparecen en la cadena s."""
3     contador = 0
4     for letra in s:
5         if letra == "A":
6             contador = contador + 1
7     return contador
```

6. Prueba

```
>>> contarA("banana")
0
>>> contarA("Ana")
1
```

```
>>> contarA("lAn")
1
>>> contarA("lAAn")
2
>>> contarA("lAnA")
2
```

Sabías que...

La instrucción `contador = contador + 1` puede reemplazarse por `contador += 1`.

En general, la mayoría de los operadores tienen versiones abreviadas para cuando la variable que queremos asignar es la misma que el primer operando:

Asignación	Asignación abreviada
`x = x + n`	`x += n`
`x = x - n`	`x -= n`
`x = x * n`	`x *= n`
`x = x / n`	`x /= n`

7.5 Darle formato a las cadenas

Muchas veces es necesario darle un formato determinado a las cadenas, o dicho de otro modo, procesar los datos de entrada para que las cadenas resultantes se vean de una manera en particular. Además, separar el formato del texto de los datos a mostrar nos permite enfocarnos en la presentación cuando eso es lo que queremos.

Por ejemplo, si queremos saludar al usuario y mostrarle la cantidad de mensajes sin leer, y en el estado tenemos `nombre` → `'Veronika'` y `no_leidos` → `8`, podemos hacer algo como:

```
>>> "Hola, " + nombre + ", tenés " + str(no_leidos) + " mensajes sin leer"
'Hola Veronika, tenés 8 mensajes sin leer'
```

Pero también podemos obtener el mismo resultado utilizando una *cadena de formato* y la función `format`:

```
>>> "Hola {}, tenés {} mensajes sin leer".format(nombre, no_leidos)
'Hola Veronika, tenés 8 mensajes sin leer'
```

La función `format` devuelve la cadena resultante de reemplazar en la cadena de formato todas las marcas `{}` por los valores indicados, convirtiendo los valores automáticamente a cadenas de texto.

Sabías que...

La función `format` se utiliza con una sintaxis que hasta ahora no habíamos visto: `cadena.format(v1, v2)` en lugar de la notación usual `format(cadena, v1, v2)`.

Por ahora es suficiente con entender que en la llamada `cadena.format(v1, v2)`, es como si la función `format` recibiera tres parámetros: `cadena`, `v1` y `v2`.

Al usar una cadena de formato de esta manera, podemos ver claramente cuál es el contenido del mensaje, evitando errores con respecto a espacios de más o de menos, interrupciones en el texto que complican su lectura, etc.

En el caso de los valores numéricos, es posible modificar la forma en la que el número es presentado. Por ejemplo, si se trata de un monto monetario, usualmente queremos mostrarlo con dos dígitos decimales, para ello utilizaremos la marca `{:.2}` para indicar dos dígitos luego del separador decimal.

```
>>> precio = 205.5
>>> 'Sin IVA: ${:.2}. Con IVA: ${:.2}'.format(precio, precio * 1.21)
'Sin IVA: $205.50. Con IVA: $248.66'
```

En otras situaciones, como el caso de un valor en un estudio médico, podemos querer mostrar el número en notación científica. En este caso utilizaremos la marca `{:.1e}`, indicando que queremos un dígito significativo luego del separador decimal.

```
>>> rojos = 4640000
>>> 'Glóbulos rojos: {:.1e}/uL'.format(rojos)
'Glóbulos rojos: 4.6e+06/uL'
```

Sabías que...

En la versión 3.7 de Python se introdujo una sintaxis nueva para generar cadenas de formato, anteponiendo la cadena con la letra `f`. Las tres siguientes expresiones son equivalentes a los ejemplos mostrados con la función `format`:

```
>>> f"Hola {nombre}, tenés {no_leidos} mensajes sin leer"
'Hola Veronika, tenés 8 mensajes sin leer'
>>> f'Sin IVA: ${precio:.2}. Con IVA: ${precio * 1.21:.2}'
'Sin IVA: $205.50. Con IVA: $248.66'
```

7.6 Resumen

Las cadenas de caracteres nos sirven para operar con todo tipo de textos. Contamos con funciones para ver su longitud, sus elementos uno a uno, o por segmentos, comparar estos elementos con otros, etc.

Referencia Python

`len(cadena)`

Devuelve el largo de una cadena, 0 si se trata de una cadena vacía.

`for caracter in cadena`

Permite realizar una acción para cada una de las letras de una cadena.

`subcadena in cadena`

Evalúa a `True` si la `cadena` contiene a la `subcadena`.

`cadena[i]`

Corresponde al valor de la cadena en la posición `i`, comenzando desde 0.
Si se utilizan números negativos, se puede acceder a los elementos desde el último (`-1`) hasta el primero (`-len(cadena)`).

`cadena[i:j]`

Permite obtener un segmento de la cadena, desde la posición `i` inclusive, hasta la posición `j` exclusive.
En el caso de que se omita `i`, se asume `0`. En el caso de que se omita `j`, se asume `len(cadena)`. Si se omiten ambos, se obtiene la cadena completa.

`cadena.isdigit()`

Devuelve `True` si todos los caracteres de la cadena son dígitos, `False` en caso contrario.

`cadena.isalpha()`

Devuelve `True` si todos los caracteres de la cadena son alfabéticos, `False` en caso contrario.

`cadena.isalnum()`

Devuelve `True` si todos los caracteres de la cadena son alfanuméricos, `False` en caso contrario.

`cadena.capitalize()`

Devuelve `True` si todos los caracteres de la cadena son alfanuméricos, `False` en caso contrario.

`cadena.upper()`

Devuelve una copia de la cadena convertida a mayúsculas.

`cadena.lower()`

Devuelve una copia de la cadena convertida a minúsculas.

Unidad 8

Listas

Los lenguajes de programación cuentan con una gran variedad de tipos de datos que permiten representar la información en función del problema a resolver. En esta unidad, se estudian las listas en Python, una estructura de datos muy útil y sencilla que se utiliza cuando se quiere agrupar elementos.

Usaremos listas para poder modelar datos compuestos, pero cuya cantidad y valor varían a lo largo del tiempo. Son secuencias *mutables* y vienen dotadas de una variedad de operaciones muy útiles.

8.1 Concepto

Formalmente, las listas son un tipo de datos compuesto, también denominada estructura de datos, que permite agrupar varios datos simples en la misma variable. Las listas poseen cuatro características:

- **Dinámicas:** La cantidad de elementos de una lista puede modificarse en tiempo de ejecución. Inicialmente, puede definirse una lista vacía o bien puede ser creada a partir de un conjunto inicial de elementos.
- **Heterogéneas:** Los elementos que forman parte de una lista pueden ser de diferentes tipos de datos. No obstante, en muchos casos vamos a trabajar con listas homogéneas (todos sus datos son del mismo tipo, por ejemplo números).
- **Indexadas:** Es posible acceder a cada uno de los elementos de una lista a través de un índice. El primer valor del índice (posición) de una lista en Python comienza en 0.
- **Mutables:** A diferencia de las cadenas de caracteres, las listas son mutables. Esto quiere decir que los elementos de una lista pueden modificarse.

La notación para las listas en Python consiste en una secuencia de valores encerrados entre corchetes y separados por comas. Por ejemplo, si representamos a los alumnos mediante su número de legajo, se puede tener una lista de inscriptos en la materia como la siguiente: `[78455, 89211, 66540, 45750]`.

Al abrirse la inscripción, antes de que hubiera inscriptos, la lista de inscriptos se representará por una lista vacía: `[]`.

En Python el tipo de dato asociado a las listas se llama `list`:

```
>>> type([78455, 89211, 66540, 45750])
<class 'list'>
```

8.2 Operaciones básicas con listas

8.2.1 Longitud de la lista. Elementos y segmentos de listas

- Como a las cadenas de caracteres ya vistas, a las listas también se les puede aplicar la función `len()` para conocer su longitud.
- Para acceder a los distintos elementos de la lista se utilizará la misma notación de índices que para las cadenas, con valores que van de 0 a la longitud de la lista -1.

```
>>> legajos = [78455, 89211, 66540, 45750]
>>> legajos[0]
78455
>>> len(legajos)
4
>>> legajos[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> legajos[3]
45750
```

- Para obtener una sublista a partir de la lista original, se utiliza la notación de rangos, como veníamos haciendo.

  Para obtener la lista que contiene sólo a quién se inscribió en segundo lugar podemos escribir:

```
>>> legajos[1:2]
[89211]
```

  Para obtener la lista que contiene al segundo y tercer inscripto podemos escribir:

```
>>> legajos[1:3]
[89211, 66540]
```

  Para obtener la lista que contiene al primero y segundo inscripto podemos escribir:

```
>>> legajos[:2]
[78455, 89211]
```

8.2.2 Cómo mutar listas

Dijimos antes que las listas son secuencias mutables. Para lograr la mutabilidad, Python provee operaciones que nos permiten cambiarle valores, agregarle valores y quitarle valores.

- Para cambiar un item de una lista, se selecciona el item mediante su índice y se le asigna el nuevo valor:

```
>>> legajos[1] = 79211
>>> legajos
[78455, 79211, 66540, 45750]
```

- Para agregar un nuevo valor al final de la lista se utiliza la operación `append()`. Escribimos `legajos.append(47890)` para agregar el legajo 47890 al final de `legajos`.

```
>>> legajos.append(47890)
>>> legajos
[78455, 79211, 66540, 45750, 47890]
```

- Para insertar un nuevo valor en la posición cuyo índice es `k` (y desplazar un lugar el resto de la lista) se utiliza la operación `insert()`.

  Escribimos `legajos.insert(2, 54988)` para insertar el legajo 54988 en la tercera posición de `legajos`.

```
>>> legajos.insert(2, 54988)
>>> legajos
[78455, 79211, 54988, 66540, 45750, 47890]
```

- Las listas no controlan si se insertan elementos repetidos. Si necesitamos exigir unicidad, debemos hacerlo mediante el código de nuestros programas.

```
>>> legajos.insert(1,78455)
>>> legajos
[78455, 78455, 79211, 54988, 66540, 45750, 47890]
```

- Para eliminar un valor de una lista se utiliza la operación `remove()`.

  Escribimos `legajos.remove(45750)` para borrar el legajo 45750 de la lista de inscriptos:

```
>>> legajos.remove(45750)
>>> legajos
[78455, 78455, 79211, 54988, 66540, 47890]
```

  Si el valor a borrar está repetido, se borra sólo su primera aparición:

```
>>> legajos.remove(78455)
>>> legajos
[78455, 79211, 54988, 66540, 47890]
```

> **⚠ Atención**
>
> Si el valor a borrar no existe, se produce un error:
>
> ```
> >>> legajos.remove(78)
> Traceback (most recent call last):
>   File "<stdin>", line 1, in <module>
> ValueError: list.remove(x): x not in list
> ```

8.2.3 Cómo buscar dentro de las listas

Queremos poder formular dos preguntas más respecto de la lista de inscriptos:

- ¿Está la persona cuyo legajo es v inscripta en esta materia?
- ¿En qué orden se inscribió la persona cuyo legajo es v?.

Veamos qué operaciones sobre listas se pueden usar para lograr esos dos objetivos:

- Para preguntar si un valor determinado es un elemento de una lista usaremos la operación `in`:

```
>>> legajos
[78455, 79211, 54988, 66540, 47890]
>>> 78 in legajos
False
>>> 66540 in legajos
True
```

El operador `in` se puede utilizar para todas las secuencias, incluyendo listas y cadenas.

- Para averiguar la posición de un valor dentro de una lista usaremos la operación `index()`.

```
>>> legajos.index(78455)
0
>>> legajos.index(47890)
4
```

Atención

Si el valor no se encuentra en la lista, se producirá un error:

```
>>> legajos.index(78)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.index(x): x not in list
```

Si el valor está repetido, el índice que devuelve es el de la primera aparición:

```
>>> [10, 20, 10].index(10)
0
```

La función `index` también se puede utilizar con cadenas.

- Para iterar sobre todos los elementos de una lista usaremos una construcción `for`:

```
>>> for p in legajos:
...     print(p)
...
78455
79211
54988
66540
47890
```

El ciclo `for <variable> in <secuencia>:` se puede utilizar sobre cualquier secuencia, incluyendo cadenas.

- Muchas veces, dentro del cuerpo del ciclo `for` es necesario contar con la posición de cada elemento de la lista. Para esto es posible utilizar la función `enumerate`:

```
>>> for i, p in enumerate(legajos):
...     print(i, p)
...
0 78455
1 79211
2 54988
3 66540
4 47890
```

Sabías que...

En Python, las listas y las cadenas son parte del conjunto de las *secuencias*. Todas las secuencias cuentan con las siguientes operaciones:

Operación	Resultado
`x in s`	Indica si el valor `x` se encuentra en `s`
`s + t`	Concantena las secuencias `s` y `t`
`s * n`	Concatena `n` copias de `s`
`s[i]`	Elemento `i` de `s`, empezando por 0
`s[i:j]`	Porción de la secuencia `s` desde `i` hasta `j` (no inclusive)
`s[i:j:k]`	Porción de la secuencia `s` desde `i` hasta `j` (no inclusive), con paso `k`
`len(s)`	Cantidad de elementos de la secuencia `s`
`min(s)`	Mínimo elemento de la secuencia `s`
`max(s)`	Máximo elemento de la secuencia `s`
`sum(s)`	Suma de los elementos de la secuencia `s`
`enumerate(s)`	Enumerar los elementos de `s` junto con sus posiciones

Además, es posible crear una lista a partir de cualquier otra secuencia, utilizando las función `list`:

```
>>> list("Hola")
['H', 'o', 'l', 'a']
```

Problema 8.1. Queremos escribir un programa que nos permita armar la lista de los inscriptos de una materia.

1. **Análisis:** El usuario ingresa datos de legajos que se van guardando en una lista.
2. **Especificación:** El programa solicitará al usuario que ingrese uno a uno los legajos de los inscriptos. Con esos números construirá una lista, que al final se mostrará.
3. **Diseño:**
   - ¿Qué estructura tiene este programa? ¿Se parece a algo conocido?

Es claramente un ciclo en el cual se le pide al usuario que ingrese uno a uno los legajos de los inscriptos, y estos números se agregan a una lista. Y en algún momento, cuando se terminaron los inscriptos, el usuario deja de cargar.

- ¿El ciclo es definido o indefinido?

  Para que fuera un ciclo definido deberíamos contar de antemano cuántos inscriptos tenemos, y luego cargar exactamente esa cantidad, pero eso no parece muy útil.

  Estamos frente a una situación parecida al problema de la lectura de los números, en el sentido de que no sabemos cuántos elementos queremos cargar de antemano. Para ese problema, en la sección 5.4, vimos una solución muy sencilla y cómoda: se le piden datos al usuario y, cuando se cargaron todos los datos se ingresa un valor arbitrario (que se usa sólo para indicar que no hay más información). A ese diseño lo hemos llamado ciclo con centinela y tiene el siguiente esquema:

```
Repetir indefinidamente:
    Pedir datos
    Si el dato pedido coincide con el centinela:
        Salir del ciclo
    Realizar cálculos
```

  Como sabemos que los números de legajo son siempre enteros positivos, podemos considerar que el centinela puede ser cualquier número menor o igual a cero. También sabemos que en nuestro caso tenemos que ir armando una lista que inicialmente no tiene ningún inscripto.

  Modificamos el esquema anterior para ajustarnos a nuestra situación:

```
La lista de inscriptos es vacía
Repetir indefinidamente:
    Pedir padrón
    Si el padrón no es positivo:
        Salir del ciclo
    Agregar el padrón a la lista
Devolver la lista de inscriptos
```

4. **Implementación:** De acuerdo a lo diseñado, el programa quedaría como se muestra en el Código 8.1.

   Para entender mejor la implementación propuesta, es una buena idea repasar los conceptos de ciclos.

5. **Prueba:** Para probarlo lo ejecutamos con algunos lotes de prueba (inscripción de tres alumnos, inscripción de cero alumnos, inscripción de alumnos repetidos):

```
$ python3 inscripcion.py
Inscripciones al curso de Introducción a la Programación
Ingresá un legajo (<=0 para terminar): 30
Ingresá un legajo (<=0 para terminar): 40
Ingresá un legajo (<=0 para terminar): 50
Ingresá un legajo (<=0 para terminar): 0
La lista de inscriptos es: [30, 40, 50]

$ python3 inscripcion.py
Inscripciones al curso de Introducción a la Programación
```

Código 8.1 `inscripcion.py`**:** Permite ingresar legajos de alumnos inscriptos

```
def inscribir_alumnos():
    """Permite inscribir alumnos al curso"""

    print("Inscripciones al curso de Introducción a la Programación")
    inscriptos = []
    finaliza_inscripcion = False
    while not(finaliza_inscripcion):
        legajo = int(input("Ingresá un legajo (<=0 para terminar): "))
        if legajo <= 0:
            finaliza_inscripcion = True
        else:
            inscriptos.append(legajo)
    return inscriptos

inscriptos = inscribir_alumnos()
print("La lista de inscriptos es:", inscriptos)
```

```
Ingresá un legajo (<=0 para terminar): 0
La lista de inscriptos es: []

$ python3 inscripcion.py
Inscripciones al curso de Introducción a la Programación
Ingresá un legajo (<=0 para terminar): 30
Ingresá un legajo (<=0 para terminar): 40
Ingresá un legajo (<=0 para terminar): 40
Ingresá un legajo (<=0 para terminar): 30
Ingresá un legajo (<=0 para terminar): 50
Ingresá un legajo (<=0 para terminar): 0
La lista de inscriptos es: [30, 40, 40, 30, 50]
```

Evidentemente el programa funciona de acuerdo a lo especificado, pero hay algo que no tuvimos en cuenta: permite inscribir a una misma persona más de una vez.

6. **Mantenimiento:** No permitir que haya legajos repetidos.
7. **Diseño revisado:** Para no permitir que haya legajos repetidos debemos revisar que no exista el legajo antes de agregarlo en la lista:

```
La lista de inscriptos es vacía
Repetir indefinidamente:
    Pedir legajo
    Si el legajo no es positivo:
        Salir del ciclo
    Si el legajo está en la lista:
        Avisar que el legajo ya está en la lista
    Sino está en la lista, se agrega
Devolver la lista de inscriptos
```

8. **Nueva implementación:** De acuerdo a lo diseñado en el párrafo anterior, el programa ahora quedaría como se muestra en el Código 8.2.
9. **Nueva prueba:** Para probarlo lo ejecutamos con los mismos lotes de prueba anteriores

Código 8.2 `inscripcion.py`**:** Permite ingresar legajos, sin repetir

```
def inscribir_alumnos():
    """Permite inscribir alumnos al curso"""

    print("Inscripciones al curso de Introducción a la Programación")
    inscriptos = []
    finaliza_inscripcion = False
    while not(finaliza_inscripcion):
        legajo = int(input("Ingresá un legajo (<=0 para terminar): "))
        if legajo <= 0:
            finaliza_inscripcion = True
        else:
            if legajo in inscriptos:
                print("El legajo ya está en la lista de inscriptos.")
            else:
                inscriptos.append(legajo)

    return inscriptos

inscriptos = inscribir_alumnos()
print("La lista de inscriptos es:", inscriptos)
```

(inscripción de tres alumnos, inscripción de cero alumnos, inscripción de alumnos repetidos):

```
$ python3 inscripcion.py
Inscripciones al curso de Introducción a la Programación
Ingresá un legajo (<=0 para terminar): 30
Ingresá un legajo (<=0 para terminar): 40
Ingresá un legajo (<=0 para terminar): 50
Ingresá un legajo (<=0 para terminar): 0
La lista de inscriptos es: [30, 40, 50]

$ python3 inscripcion.py
Inscripciones al curso de Introducción a la Programación
Ingresá un legajo (<=0 para terminar): 0
La lista de inscriptos es: []

$ python3 inscripcion.py
Inscripciones al curso de Introducción a la Programación
Ingresá un legajo (<=0 para terminar): 30
Ingresá un legajo (<=0 para terminar): 40
Ingresá un legajo (<=0 para terminar): 40
El legajo ya está en la lista de inscriptos.
Ingresá un legajo (<=0 para terminar): 30
El legajo ya está en la lista de inscriptos.
Ingresá un legajo (<=0 para terminar): 50
Ingresá un legajo (<=0 para terminar): 0
La lista de inscriptos es: [30, 40, 50]
```

Ahora el resultado es satisfactorio: no tenemos inscriptos repetidos.

Ahora bien, ¿Qué deberíamos hacer si queremos guardar el apellido y nombre del estu-

diante en una variable, además de su legajo?

10. **Mantenimiento:** Permitir guardar el nombre y apellido de los estudiantes.

11. **Diseño revisado:** Para no permitir ingresar el nombre y apellido de los estudiantes, es posible utilizar dos listas: una para los legajos y otra para los nombres y apellidos. ¿Cómo sabremos a que legajo de una lista le corresponde el apellido y nombre de la otra? Sencillo, por el orden en que aparecen en las estructuras de datos.

```
La lista de inscriptos es vacía
Repetir indefinidamente:
    Pedir legajo
    Si el legajo no es positivo:
        Salir del ciclo
    Si el legajo está en la lista:
        Avisar que el legajo ya está en la lista
    Sino está en la lista, se agrega el legajo y se solicita el nombre y
    ↪ apellido
Devolver la lista de inscriptos
```

12. **Nueva implementación:** De acuerdo a lo diseñado en el párrafo anterior, el programa ahora quedaría como se muestra en el Código 8.3.

Código 8.3 `inscripcion.py`**:** Permite ingresar legajos, sin repetir, con el apellido y nombre de los estudiantes

```python
def inscribir_alumnos():
    """Permite inscribir alumnos al curso"""

    print("Inscripciones al curso de Introducción a la Programación")
    inscriptos_legajo = []
    inscriptos_apellido_y_nombre = []
    finaliza_inscripcion = False
    while not(finaliza_inscripcion):
        legajo = int(input("Ingresá un legajo (<=0 para terminar): "))
        if legajo <= 0:
            finaliza_inscripcion = True
        else:
            if legajo in inscriptos_legajo:
                print("El legajo ya está en la lista de inscriptos.")
            else:
                inscriptos_legajo.append(legajo)
                apellido_nombre = input("Apellido y nombre: ")
                inscriptos_apellido_y_nombre.append(apellido_nombre)

    return inscriptos_legajo, inscriptos_apellido_y_nombre

legajos, apellidos_y_nombres = inscribir_alumnos()

print("Listado de Inscriptos FINAL:")
for i in range(len(legajos)):
    print(f"Inscripto {i}: ({legajos[i]}) {apellidos_y_nombres[i]}")
```

13. **Nueva prueba:** Para probarlo lo ejecutamos nuevamente simulando la inscripción de

tres alumnos:

```
$ python3 inscripcion.py
Inscripciones al curso de Introducción a la Programación
Ingresá un legajo (<=0 para terminar): 30
Apellido y nombre: Hatake Kakashi

Ingresá un legajo (<=0 para terminar): 40
Apellido y nombre: Uzumaki Naruto

Ingresá un legajo (<=0 para terminar): 50
Apellido y nombre: Uchiha Itachi

Ingresá un legajo (<=0 para terminar): 0

Listado de Inscriptos FINAL:
Inscripto 0: (30) Hatake Kakashi
Inscripto 1: (40) Uzumaki Naruto
Inscripto 2: (50) Uchiha Itachi
```

Ahora el resultado es satisfactorio: no tenemos inscriptos repetidos y podemos guardar el nombre y apellido.

8.3 Ordenar listas

Nos puede interesar que los elementos de una lista estén ordenados: una vez que finalizó la inscripción en un curso, tener a los legajos de los alumnos por orden de inscripción puede ser muy incómodo, siempre será preferible tenerlos ordenados por número para realizar cualquier comprobación.

Python provee dos operaciones para obtener una lista ordenada a partir de una lista desordenada.

- Para dejar la lista original intacta pero obtener una nueva lista ordenada a partir de ella, se usa la función `sorted`.

```
>>> bs = [5, 2, 4, 2]
>>> cs = sorted(bs)
>>> bs
[5, 2, 4, 2]
>>> cs
[2, 2, 4, 5]
```

- Para modificar directamente la lista original usaremos la operación `sort()`.

```
>>> ds = [5, 3, 4, 5]
>>> ds.sort()
>>> ds
[3, 4, 5, 5]
```

8.4 Listas y cadenas

A partir de una cadena de caracteres, podemos obtener una lista con sus componentes usando la función `split`.

Si queremos obtener las palabras (separadas entre sí por espacios) que componen la cadena `legajos` escribiremos simplemente `legajos.split()`:

```
>>> c = "   Una    cadena   con    espacios    "
>>> c.split()
['Una', 'cadena', 'con', 'espacios']
```

En este caso `split` elimina todos los blancos de más, y devuelve sólo las palabras que conforman la cadena.

Si en cambio el separador es otro carácter (por ejemplo la arroba, `"@"`), se lo debemos pasar como parámetro a la función `split`. En ese caso se considera una componente todo lo que se encuentra entre dos arrobas consecutivas. En el caso particular de que el texto contenga dos arrobas una a continuación de la otra, se devolverá una componente vacía:

```
>>> d="@@Una@@@cadena@@@con@@arrobas@"
>>> d.split("@")
['', '', 'Una', '', '', 'cadena', '', '', 'con', '', 'arrobas', '']
```

La "casi"–inversa de `split` es una función `join` que tiene la siguiente sintaxis:

```
<separador>.join(<lista de componentes a unir>)
```

y que devuelve la cadena que resulta de unir todas las componentes separadas entre sí por medio del *separador*:

```
>>> xs = ['aaa', 'bbb', 'cccc']
>>> " ".join(xs)
'aaa bbb cccc'
>>> ", ".join(xs)
'aaa, bbb, cccc'
>>> "@@".join(xs)
'aaa@@bbb@@cccc'
```

Referencia Python

`[valor1, valor2, valor3]`

Las listas se definen como una sucesión de valores encerrados entre corchetes y separados por comas. Se les puede agregar, quitar o cambiar los valores que contienen.

```
lista = [1, 2, 3]
lista[0] = 5
```

Caso particular:

```
lista_vacia = []
```

`x, y, z = secuencia`

Es posible *desempaquetar* una secuencia, asignando a la izquierda tantas variables como elementos tenga la secuencia. Cada variable tomará el valor del elemento que se encuentra en la misma posición.

`len(secuencia)`

Devuelve la cantidad de elementos que contiene la secuencia, 0 si está vacía.

`for elemento in secuencia:`

Itera uno a uno por los elementos de la secuencia.

`elemento in secuencia`

Indica si el elemento se encuentra o no en la secuencia

`secuencia[i]`

Corresponde al valor de la secuencia en la posición `i`, comenzando desde 0.

Si se utilizan números negativos, se puede acceder a los elementos desde el último (`-1`) hasta el primero (`-len(secuencia)`).

En el caso de las tuplas o cadenas (inmutables) sólo puede usarse para obtener el valor, mientra que en las listas (mutables) puede usarse también para modificar su valor.

`secuencia[i:j:k]`

Permite obtener un segmento de la secuencia, desde la posición `i` inclusive, hasta la posición `j` exclusive, con paso `k`.

En el caso de que se omita `i`, se asume `0`. En el caso de que se omita `j`, se asume `len(secuencia)`. En el caso de que se omita `k`, se asume 1. Si se omiten todos, se obtiene una copia completa de la secuencia.

`lista.append(valor)`

Agrega un elemento al final de la lista.

`lista.insert(posicion, valor)`

Agrega un elemento a la lista, en la posición `posicion`.

`lista.remove(valor)`

Quita de la lista la primera aparción de elemento, si se encuentra. De no encontrarse en la lista, se produce un error.

`lista.pop()`

Quita el elemento del final de la lista, y lo devuelve. Si la lista está vacía, se produce un error.

`lista.pop(posicion)`

Quita el elemento que está en la posición indicada, y lo devuelve. Si la lista tiene menos de `posicion + 1` elementos, se produce un error.

`lista.index(valor)`

Devuelve la posición de la primera aparición de valor. Si no se encuentra en la lista, se produce un error.

`sorted(secuencia)`

Devuelve una lista nueva, con los elementos de la secuencia ordenados.

`lista.sort()`

Ordena la misma lista.

`cadena.split(separador)`

Devuelve una lista con los elementos de cadena, utilizando `separador` como separador de elementos.

Si se omite el separador, toma todos los espacios en blanco como separadores.

`separador.join(lista)`

Genera una cadena a partir de los elementos de `lista`, utilizando `separador` como unión entre cada elemento y el siguiente.

8.5 Búsqueda en listas

El problema de la búsqueda

Presentamos ahora uno de los problemas clásicos de la computación, *el problema de la búsqueda*, que se puede enunciar de la siguiente manera:

Problema: Dada una lista L y un valor x devolver el índice de x en L si x está en L, y -1 si x no está en L.

Este problema tiene una solución muy sencilla en Python: se puede usar directamente la poderosa función `index()` de lista.

Probamos esa solución para ver qué pasa:

```
>>> [1, 3, 5, 7].index(5)
2
>>> [1, 3, 5, 7].index(20)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.index(x): x not in list
```

Vemos que usar la función `index()` resuelve nuestro problema si el valor buscado está en la lista, pero si el valor no está no sólo no devuelve un -1, sino que se produce un error.

El problema es que para poder aplicar la función `index()` debemos estar seguros de que el valor está en la lista, y para averiguar eso Python nos provee del operador `in`:

```
>>> 5 in [1, 3, 5, 7]
True
>>> 20 in [1, 3, 5, 7]
False
```

O sea que si llamamos a la función `index()` sólo cuando el resultado de `in` es verdadero, y devolvemos -1 cuando el resultado de `in` es falso, estaremos resolviendo el problema planteado usando sólo funciones provistas por Python. La solución se plantea a continuación:

```
def busqueda_con_index(L, x):
    """Busca un elemento x en una lista L.

    Si x está en L devuelve el índice,
    de lo contrario devuelve -1.
    """
    resultado = -1
    if x in L:
        resultado = L.index(x)

    return resultado
```

Probamos la función `busqueda_con_index()`:

```
>>> busqueda_con_index([1, 4, 54, 3, 0, -1], 1)
0
>>> busqueda_con_index([1, 4, 54, 3, 0, -1], -1)
5
>>> busqueda_con_index([1, 4, 54, 3, 0, -1], 3)
3
>>> busqueda_con_index([1, 4, 54, 3, 0, -1], 44)
-1
>>> busqueda_con_index([], 0)
-1
```

¿Cuántas comparaciones hace este programa?

Es decir, ¿cuánto esfuerzo computacional requiere este programa? ¿Cuántas veces compara el valor que buscamos con los datos de la lista? No lo sabemos porque no sabemos cómo están implementadas las operaciones `in` e `index()`. La pregunta queda planteada por ahora pero daremos un método para averiguarlo más adelante en esta unidad.

8.5.1 Búsqueda lineal

Nos interesa ver qué sucede si programamos la búsqueda usando operaciones más elementales, y no las grandes primitivas `in` e `index()`. Esto nos permitirá estudiar una solución que pueda trasladarse a otros lenguajes de programación.

Supongamos entonces que en nuestra versión de Python no existen ni `in` ni `index()`. Podemos en cambio acceder a cada uno de los elementos de la lista a través de una construcción `for`, y también, por supuesto, podemos acceder a un elemento de la lista mediante un índice.

Diseñamos la siguiente solución: podemos comparar uno a uno los elementos de la lista con el valor de `x`, y retornar el valor de la posición donde lo encontramos, en caso de encontrarlo. Si llegamos al final de la lista sin haber encontrado el valor es porque el valor de `x` no está en la lista, y en ese caso retornamos -1.

En esta solución necesitamos una variable `i` que cuente en cada momento en qué posición de la lista estamos parados. Esta variable se inicializa en 0 antes de entrar en el ciclo y se incrementa en 1 en cada paso.

El programa nos queda entonces como se muestra a continuación:

```
def busqueda_lineal(lista, x):
    """Si x está en lista devuelve su posición en lista, de lo
    contrario devuelve -1.
    """
    resultado = -1
    i = 0
    for z in lista: ❶
        if z == x: ❷
            resultado = i ❸
        i += 1
    return resultado
```

Y ahora lo probamos:

```
>>> busqueda_lineal([1, 4, 54, 3, 0, -1], 44)
-1
>>> busqueda_lineal([1, 4, 54, 3, 0, -1], 3)
3
>>> busqueda_lineal([1, 4, 54, 3, 0, -1], 0)
4
>>> busqueda_lineal([], 42)
-1
```

¿Cuántas comparaciones hace este programa?

Volvemos a preguntarnos lo mismo que en la sección anterior, pero con el nuevo programa: ¿cuánto esfuerzo computacional requiere este programa?, ¿cuántas veces compara el valor que buscamos con los datos de la lista? Ahora podemos analizar el código de `busqueda\_lineal`:

- La línea ❶ es un ciclo que recorre uno a uno los elementos de la lista, y en el cuerpo de ese ciclo, en ❷ se compara cada elemento con el valor buscado. En el caso de encontrarlo (❸) guarda la posición.
- Independientemente de que el valor esté o no esté en la lista, se recorrerá la lista entera, haciendo una comparación por cada elemento.

¿Hay alguna forma de modificar la solución para que las comparaciones sean potencialmente menores? Hay varias formas de lograr esto. Vamos a centrarnos en una, la cual consiste en cambiar la estructura iterativa `for` por un `while`. Exploremos esta posibilidad:

```
def busqueda_lineal_2(lista, x):
    """Si x está en lista devuelve su posición en lista, de lo
    contrario devuelve -1.
    """
    resultado = -1
    i = 0
    while (i<len(lista) and resultado==-1): ❹
        if lista[i] == x:
            resultado = i
        i += 1

    return resultado
```

El mayor cambio está en la instrucción marcada con ❹, donde modificamos el ciclo `for` por un ciclo indefinido, el cual va a iterar siempre que no se haya encontrado el valor buscado y además el valor de i sea menor que el tamaño de la lista. De esta manera, si el valor está en la posición p de la lista se hacen p comparaciones. En el *peor caso*, si el valor no está, se hacen tantas comparaciones como elementos tenga la lista.

Nuestra hipótesis es: **Si la lista crece, la cantidad de comparaciones para encontrar un valor arbitrario crecerá en forma proporcional al tamaño de la lista**. Por lo tanto diremos que:

> El algoritmo de búsqueda lineal tiene un comportamiento *proporcional a la longitud de la lista involucrada*, o que es un algoritmo *lineal*.

8.5.2 Búsqueda binaria

Si podemos suponer que la lista está previamente ordenada, ¿podemos encontrar una manera más eficiente de buscar elementos sobre ella?

En principio hay una modificación muy simple que podemos hacer sobre el algoritmo de búsqueda lineal: si estamos buscando el elemento x en una lista que está ordenada de menor a mayor, en cuanto encontremos algún elemento mayor a x podemos estar seguros de que x no está en la lista, por lo que no es necesario continuar recorriendo el resto.

¿Podemos hacer algo mejor? Trataremos de aprovechar el hecho de que la lista está ordenada y vamos a hacer algo distinto: nuestro espacio de búsqueda se irá achicando a segmentos cada vez menores de la lista original.

La idea es descartar segmentos de la lista donde el valor seguro que no puede estar:

1. Consideramos como segmento inicial de búsqueda a la lista completa.
2. Analizamos el punto medio del segmento (el valor central); si es el valor buscado, devolvemos el índice del punto medio.

3. Si el valor central es mayor al buscado, podemos descartar el segmento que está desde el punto medio hacia la a derecha.
4. Si el valor central es menor al buscado, podemos descartar el segmento que está desde el punto medio hacia la izquierda.
5. Una vez descartado el segmento que no nos interesa, volvemos a analizar el segmento restante, de la misma forma.
6. Si en algún momento el segmento a analizar tiene longitud 0 o negativa significa que el valor buscado no se encuentra en la lista.

Para señalar la porción del segmento que se está analizando a cada paso, utilizaremos dos variables (`izq` y `der`) que contienen la posición de inicio y la posición de fin del segmento que se está considerando. De la misma manera usaremos la varible `medio` para contener la posición del punto medio del segmento.

En la Figura 8.1 vemos qué pasa cuando se busca el valor 18 en la lista [1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23].

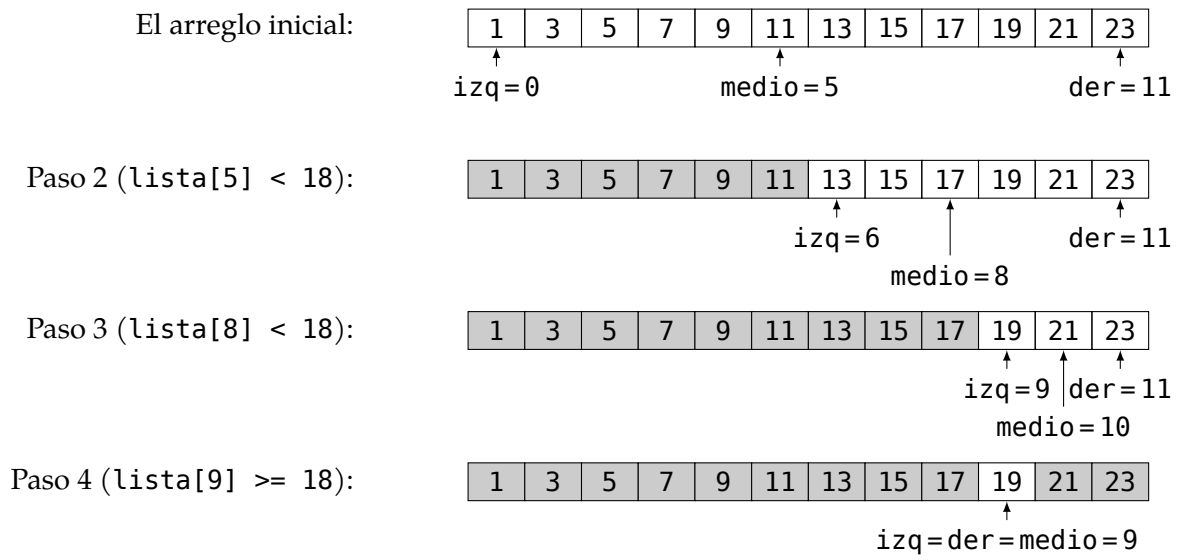


Figura 8.1: Ejemplo de una búsqueda usando el algoritmo de búsqueda binaria. Como no se encontró al valor buscado, devuelve -1.

En el Código 8.4 mostramos una posible implementación de este algoritmo, incluyendo una instrucción de depuración con `print` para verificar su funcionamiento.

Código 8.4 `busqueda_binaria.py`**:** Función de búsqueda binaria

```
def busqueda_binaria(lista, x):
    """Búsqueda binaria

    Precondición: la lista está ordenada
    Devuelve -1 si x no está en lista;
    Devuelve p tal que lista[p] == x, si x está en lista
    """

    posicion = -1
    encontrado = False

    izq = 0
    der = len(lista) - 1

    while izq <= der and not(encontrado):
        medio = (izq + der) // 2
        print("[DEBUG]", "izq:", izq, "der:", der, "medio:", medio)

        if lista[medio] == x:
            # Encontramos el elemento; devolvemos su posición
            encontrado = True
            posicion = medio

        if lista[medio] > x:
            # Seguimos buscando en el segmento de la izquierda
            der = medio - 1
        else:
            # Seguimos buscando en el segmento de la derecha
            izq = medio + 1

    return posicion
```

A continuación varias ejecuciones de prueba:

```
>>> busqueda_binaria([1, 3, 5], 0)
[DEBUG] izq: 0 der: 2 medio: 1
[DEBUG] izq: 0 der: 0 medio: 0
-1
>>> busqueda_binaria([1, 3, 5], 1)
[DEBUG] izq: 0 der: 2 medio: 1
[DEBUG] izq: 0 der: 0 medio: 0
0
>>> busqueda_binaria([1, 3, 5], 2)
[DEBUG] izq: 0 der: 2 medio: 1
[DEBUG] izq: 0 der: 0 medio: 0
-1
>>> busqueda_binaria([1, 3, 5], 3)
[DEBUG] izq: 0 der: 2 medio: 1
1
>>> busqueda_binaria([1, 3, 5], 5)
[DEBUG] izq: 0 der: 2 medio: 1
[DEBUG] izq: 2 der: 2 medio: 2
```

```
2
>>> busqueda_binaria([1, 3, 5], 6)
[DEBUG] izq: 0 der: 2 medio: 1
[DEBUG] izq: 2 der: 2 medio: 2
-1
>>> busqueda_binaria([1], 1)
[DEBUG] izq: 0 der: 0 medio: 0
0
```

¿Cuántas comparaciones hace este programa?

Para responder esto pensemos en el peor caso, es decir, que se descartaron varias veces partes del segmento para finalmente llegar a un segmento vacío y el valor buscado no se encontraba en la lista.

En cada paso el segmento se divide por la mitad y se desecha una de esas mitades, y en cada paso se hace una comparación con el valor buscado. Por lo tanto, la cantidad de comparaciones que hacen con el valor buscado es aproximadamente igual a la cantidad de pasos necesarios para llegar a un segmento de tamaño 1. Veamos el caso más sencillo para razonar, y supongamos que la longitud de la lista es una potencia de 2, es decir `len(lista)` $= 2^k$:

- Luego del primer paso, el segmento a tratar es de tamaño 2^k.
- Luego del segundo paso, el segmento a tratar es de tamaño 2^{k-1}.
- Luego del tercer paso, el segmento a tratar es de tamaño 2^{k-2}.

  ...
- Luego del paso k, el segmento a tratar es de tamaño $2^{k-k} = 1$.

Por lo tanto este programa hace aproximadamente k comparaciones con el valor buscado cuando `len(lista)` $= 2^k$. Pero si despejamos k de la ecuación anterior, podemos ver que este programa realiza aproximadamente $\log_2($`len(lista)`$)$ comparaciones.

Cuando `len(lista)` no es una potencia de 2 el razonamiento es menos prolijo, pero también vale que este programa realiza aproximadamente $\log_2($`len(lista)`$)$ comparaciones. Concluimos entonces que:

> Si podemos suponer que la lista está previamente ordenada, podemos utilizar el algoritmo de búsqueda binaria, cuyo comportamiento es proporcional al *logaritmo* de la cantidad de elementos de la lista, y por lo tanto *muchísimo* más eficiente que la búsqueda lineal.

Veamos un ejemplo para entender cuánto más eficiente es la búsqueda binaria. Supongamos que tenemos una lista de un millón de elementos.

- El algoritmo de búsqueda lineal hará una cantidad de operaciones proporcional a un millón; es decir que en el peor caso hará 1 000 000 comparaciones, y en un caso promedio, 500 000 comparaciones.
- El algoritmo de búsqueda binaria hará como máximo $\log_2(1\,000\,000)$ comparaciones, o sea ¡no más que 20 comparaciones!.

Sabías que...

Para que el algoritmo de búsqueda binaria sea eficiente hay un requisito adicional, que nosotros dimos por sentado: si la lista `L` tiene N elementos, el esfuerzo computacional para evaluar `L[i]` debe ser el mismo *sin importar el valor de N o de* `i`.

Por ejemplo, si la lista tiene un millón de elementos, el esfuerzo computacional de evaluar `L[0]` debe ser el mismo que para evaluar `L[500000]` o `L[999999]`; y el mismo que si la lista tuviera 10 millones o 100 millones. Cuando esto ocurre, decimos que la operación `L[i]` es de *tiempo constante*.

La implementación interna de las listas y cadenas en Python garantiza que se cumple esta condición.

8.6 Resumen

- Todos los lenguajes de programación proveen varias estructuras que nos permiten agrupar los datos que tenemos. Python nos provee las **listas**, que son estructuras **mutables** que permiten agrupar valores, con la posibilidad de agregar, quitar o reemplazar sus elementos.
- Las listas se utilizan en las situaciones en las que los elementos a agrupar pueden ir variando a lo largo del tiempo. Por ejemplo, para representar un las notas de un alumno en diversas materias, los inscriptos para un evento o la clasificación de los equipos en una competencia.
- Las cadenas y las listas son dos tipos diferentes de **secuencias**. Las secuencias ofrecen un conjunto de operaciones básicas, como obtener la longitud y recorrer sus elementos, que se aplican de la misma manera sin importar qué tipo de secuencia es.
- La **búsqueda** de un elemento en una secuencia es un algoritmo básico pero importante. El problema que intenta resolver puede plantearse de la siguiente manera: Dada una secuencia de valores y un valor, devolver el índice del valor en la secuencia, si se encuentra, de no encontrarse el valor en la secuencia señalizarlo apropiadamente.
- Una de las formas de resolver el problema es mediante la **búsqueda lineal**, que consiste en ir revisando uno a uno los elementos de la secuencia y comparándolos con el elemento a buscar. Este algoritmo no requiere que la secuencia se encuentre ordenada, la cantidad de comparaciones que realiza es proporcional a `len(secuencia)`.
- Cuando la secuencia sobre la que se quiere buscar está ordenada, se puede utilizar el algoritmo de **búsqueda binaria**. Al estar ordenada la secuencia, se puede desacartar en cada paso la mitad de los elementos, quedando entonces con una eficiencia algorítmica proporcional a log_2`(len(secuencia))`.
- El análisis del comportamiento de un algoritmo puede ser muy engañoso si se tiene en cuenta el mejor caso, por eso suele ser mucho más ilustrativo tener en cuenta el **peor caso**. En algunos casos particulares podrá ser útil tener en cuenta, además, el **caso promedio**.

Licencia y Copyright