Informática

Introducción a la Programación usando Python como herramienta

1ra edición

Prólogo

Comúnmente se entiende por "Informática" al tratamiento automático de la información. Se trata, en general, de un conjunto de conocimientos teóricos y prácticos, relativos al ámbito de la ciencia y de la tecnología, que se combinan para posibilitar el tratamiento racional y automático de la información mediante sistemas de computación. En términos educativos, se denomina "Informática" al estudio del diseño y desarrollo de los sistemas computacionales y de los ámbitos computacionales; incluye el estudio del diseño, mantenimiento, e integración de las aplicaciones de software. Es obvio que tan vasto alcance no cabe en los contenidos de una asignatura cuatrimestral, no obstante, podemos imaginar un camino para introducirnos gradualmente en ese terreno del conocimiento.

Comencemos pensando que una computadora es un dispositivo electrónico en el cual se puede guardar, extraer y procesar datos (transformar y compartir información de manera estructurada), bajo el control de "programas" que le permiten desarrollar diversas tareas. Los programas están constituidos por secuencias de instrucciones que indican a la computadora, paso a paso, qué es lo que debe hacer[1]. Esos programas almacenados en la computadora, están escritos en un lenguaje que resulta difícil de entender para el común de las personas; por esta razón, existen diversos lenguajes para desarrollar programas ("lenguajes de programación") que son traducidos al lenguaje que la máquina puede entender.

Los distintos *lenguajes de programación* tienen diversas características que los hacen más, o menos, adecuados para la resolución de determinados problemas.

En este contexto, la manera más divertida de comenzar a aprender informática, quizás sea escribiendo programas que, rápidamente, logren hacer que la computadora realice alguna tarea específica. Entonces se hace necesaria la elección de un *lenguaje de programación* para aprender el arte de programar. Pero no sólo se trata de elegir una herramienta que permita diseñar soluciones para requerimientos dados (*algoritmos* de resolución de problemas), sino que, además, el esfuerzo que demande el aprendizaje del lenguaje de programación, debiera otorgar como recompensa la posible utilización de ese lenguaje en futuros emprendimientos.

Varias cualidades requeridas en esa elección caracterizan al lenguaje Python:

- desde el punto de vista sintáctico y semántico, es lo suficientemente sencillo y fácil de aprender
- está disponible en múltiples plataformas, de manera gratuita
- permite ejecución interactiva del código, agilizando el aprendizaje
- facilita el desarrollo rápido de una aplicación, permitiendo obtener resultados inmediatos
- es potente y versátil para abordar aspectos de distintos paradigmas de programación

[1] Cuando programamos un ciclo de lavado en un lavarropas automático, sólo elegimos opciones de una programación ya predeterminada. En nuestro caso, hablamos de programar cualquier tarea computable.

- tiene una amplia difusión, que supone su permanencia en el tiempo

A las cualidades del lenguaje, podemos agregar la disponibilidad de una metodología sencilla para la resolución de problemas con programación.

En este punto, es muy importante aclarar que no se trata de un curso de lenguaje Python, sino que veremos sólo algunas de sus estructuras básicas, comunes a la mayoría de los lenguajes de programación imperativos, que nos permitan ejercitar el desarrollo de algoritmos informáticos para la resolución de problemas, mediante la práctica intensiva de la definición y utilización de "funciones".

En síntesis, creemos que introducirse en la Informática comenzando a programar con Python es una excelente oportunidad, de modo que eso es lo que haremos.

Para este propósito hemos adaptado este apunte, que es una versión reducida (y levemente modificada) de la segunda edición del libro "Algoritmos y Programación I. Aprendiendo a programar usando Python como herramienta", cuya accesibilidad agradecemos a los docentes del curso de Algoritmos y Programación I de las carreras "Ingeniería en Informática" y "Licenciatura en Análisis de Sistemas", de la Facultad de Ingeniería de la UBA.

Finalmente, queremos destacar nuestro especial agradecimiento a los autores del libro: Rosita Wachenchauzer, Margarita Manterola, Maximiliano Curia, Marcos Medrano, Nicolás Paez, Diego Essaya, Dato Simó, Sebastián Santisi.

También agradecemos, transitivamente, a los colaboradores que menciona su respectivo prólogo: Melisa Halsband, Alberto Bertogli, Pablo Antonio, Pablo Najt, Leandro Ferrigno, Martín Albarracín, Gastón Kleiman, Matías Gavinowich, Bruno Merlo Schurmann, Damián Schenkelman, Débora Martín, Ezequiel Genender, Fabricio Pontes Harsich, Fabrizio Graffe, Federico Barrios, Federico López, Gastón Martinez, Gastón Goncalves, Ignacio Garay, Javier Choque, Jennifer Woites, Manuel Soldini, Martín Buchwald, Pablo Musumeci, Agustín Santiago, Agustina Méndez, Alan Rinaldi, Alejandro Levinas, Ana Czarnitzki, Ariel Vergara, Ayelén Bibiloni Lombardi, Carlos Talavara, Constanza González, Daniela Riesgo, Daniela Soto, Diego Alfonso, Emiliano Sorbello, Eugenia Mariotti, Federico Esteban, Florencia Álvarez Etcheverry, Florencia Rodríguez, Franco Di María, Gianmarco Cafferatta, Ignacio Sueiro, Joel Saidman, Juan Costamagna, Juan Ignacio Kristal, Juan Patricio Marshall, Julián Crespo, Klaus Lungwitz, Lucas Perea, Luciano Sportelli Castro, Manuel Battan, Manuel Porto, Manuel Sturla, Martín Coll, Martín Dabat, Matías Scacosky, Maximiliano Suppes, Maximiliano Yung, Miguel Alfaro, Milena Farotto, Nicolás Poncet, Ramiro Santos, Robinson Fang, Rodrigo Vélez, Sebastián González, Sofía Morseletto, Tomás Rocchi.

Avellaneda, marzo de 2020

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 Computadoras y programas

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.

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.

Un *programa de computadora* es un conjunto de *instrucciones* paso a paso que le indican a una computadora cómo realizar una tarea dada, y en cada momento uno puede elegir ejecutar 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 especialmente 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.

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 la compra. Habrá que preguntarle a quien nos da la orden cuál es la interpretación correcta. Pero 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**, **Yahoo!**, **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**.

1.2 El mito de la máquina todopoderosa

Muchas veces la gente se imagina que con la computadora se puede hacer cualquier cosa; o que si bien hubo tareas que no eran posibles de realizar hace 50 años, sí lo serán cuando las computadoras crezcan en poder (memoria, velocidad), y se vuelvan máquinas todopoderosas.

Sin embargo eso no es así: existen algunos problemas, llamados *no computables* que nunca podrán ser resueltos por una computadora digital, por más poderosa que ésta sea. La computabilidad es la rama de la computación que se ocupa de estudiar qué tareas son computables y qué tareas no lo son.

De la mano del mito anterior, viene el mito del lenguaje todopoderoso: hay problemas que son no computables porque en realidad se utiliza algún lenguaje que no es el apropiado.

En realidad todas las computadoras pueden resolver los mismos problemas, y eso es independiente del lenguaje de programación que se use. Las soluciones a los problemas computables se pueden escribir en cualquier lenguaje de programación. Eso no significa que no haya lenguajes más adecuados que otros para la resolución de determinados problemas, pero la adecuación está relacionada con temas tales como la elegancia, la velocidad, la facilidad para describir un problema de manera simple, etc., nunca con la capacidad de resolución.

Los problemas no computables no son los únicos escollos que se le presentan a la computación. Hay otros problemas que si bien son computables demandan para su resolución un esfuerzo enorme en tiempo y en memoria. Estos problemas se llaman *intratables*. El análisis de algoritmos se ocupa de separar los problemas tratables de los intratables, encontrar la solución más barata para resolver un problema dado, y en el caso de los intratables, resolverlos de manera aproximada: no encontramos la verdadera solución porque no nos alcanzan los recursos para eso, pero encontramos una solución bastante buena y que nos insume muchos menos recursos (el orden de las respuestas de Google a una búsqueda es un buen ejemplo de una solución aproximada pero no necesariamente óptima).

En este curso trabajaremos con problemas no sólo computables sino también tratables.

Algunos ejemplos de los problemas que encararemos y de sus soluciones:

Problema 1.1. Dado un número N se quiere calcular N^{33}.

Una solución posible, por supuesto, es hacer el producto $N \cdot N \cdots N$, que involucra 32 multiplicaciones.

Otra solución, mucho más eficiente es:

- Calcular $N \cdot N$.
- Al resultado anterior mutiplicarlo por sí mismo con lo cual ya disponemos de N^4.
- Al resultado anterior mutiplicarlo por sí mismo con lo cual ya disponemos de N^8.
- Al resultado anterior mutiplicarlo por sí mismo con lo cual ya disponemos de N^{16}.
- Al resultado anterior mutiplicarlo por sí mismo con lo cual ya disponemos de N^{32}.
- Al resultado anterior mutiplicarlo por N con lo cual conseguimos el resultado deseado con sólo 6 multiplicaciones.

Cada una de estas dos soluciones representa un *algoritmo*, es decir un método de cálculo, diferente. Para un mismo problema puede haber algoritmos diferentes que lo resuelven, cada uno con un costo distinto en términos de recursos computacionales involucrados.

Sabías que...

La palabra *algoritmo* no es una variación de *logaritmo*, sino que 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, que literalmente significa: "Padre de Ja'far Mohammed, hijo de Moises, nativo de Jiva". 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".

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.

Problema 1.2. Tenemos que permitir la actualización y consulta de una guía telefónica.

Para este problema no hay una solución única: hay muchas y cada una está relacionada con un contexto de uso. ¿De qué guía estamos hablando: la guía de una pequeña oficina, un pequeño pueblo, una gran ciudad, la guía de la Argentina? Y en cada caso ¿de qué tipo de consulta estamos hablando: hay que imprimir un listado una vez por mes con la guía completa, se trata de una consulta en línea, etc.? Para cada contexto hay una solución diferente, con los datos guardados en una *estructura de datos* apropiada, y con diferentes algoritmos para la actualización y la consulta.

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 o consola (intérprete de comandos del Sistema Operativo, sea Windows o Linux).

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, de junio de 2018). 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.

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.

```
$ python3
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.
>>>
```

Para orientarnos, el intérprete muestra los símbolos `>>>` (llamaremos a esto el *prompt*), indicando que podemos escribir a continuación una *sentencia* u orden que será evaluada.

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 potenciación). La sintaxis es la convencional (valores intercalados con operaciones), y se puede usar paréntesis para modificar el orden de asociación natural de las operaciones (potenciación, 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
```

En Python tenemos dos tipos de datos numéricos: los *números enteros* y los *números de punto flotante*. Los números enteros representan el valor entero exacto que ingresemos. Los números flotantes 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 flotantes.

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, el dinero en una cuenta bancaria[1]. En cambio vamos a elegir flotantes 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, el número de Avogadro, entre otros casos.

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'
```

1.4 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] ¿Pero la moneda no tiene decimales?, ¡sí!, pero conviene representar el saldo como la cantidad total de centavos, que es un número entero, ya que es muy importante almacenar la suma exacta que hay en la cuenta.

1.5 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 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.

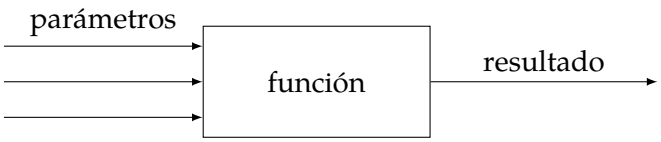


Figura 1.1: Una función recibe parámetros y devuelve un resultado.

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.

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[2] con la que se escribe la línea `return` es importante: le indica a Python que estamos

[2]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.

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:

```
>>> 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."
```

En este caso, además de indicar el nombre de la función (`hola`), debemos darle un nombre al parámetro (`alguien`), cuyo valor será reemplazado por una cadena de texto cuando se invoque a la función. Por ejemplo, podemos invocarla dos veces, para saludar a Ana y a Juan:

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

Problema 1.5.1. 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 1.5.2. 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
```

1.6 Una instrucción un poco más compleja: el ciclo definido

Problema 1.6.1. Supongamos que queremos calcular la suma de los primeros 5 números cuadrados.

Solución. Dado que ya tenemos la función `cuadrado`, 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*, 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.

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 1.6.2. 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
```

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, `cuad100.py`, que nos permite calcular la suma de los primeros 100 cuadrados.

Código 1.1 `cuad100.py`**:** Imprime la suma de los primeros 100 números cuadrados

```
1 def suma_cuadrados(n):
2     suma = 0
3     for x in range(1, n + 1):
4         suma = suma + cuadrado(x)
5     return suma
6
7 print("La suma de los primeros 100 cuadrados es", suma_cuadrados(100))
```

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
```

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 `cuad100.py` debemos llamar a la función `print` para mostrar el resultado.

Para ejecutar el programa debemos abrir una consola del sistema operativo y ejecutar `python cuad100.py`:

```
$ python cuad100.py
La suma de los primeros 100 cuadrados es 338350
```

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. Por ejemplo:

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

Solución. Ya habíamos escrito la función `hola` que nos permitía saludar a una persona si sabíamos su nombre. Pero aún no sabemos cómo obtener el nombre del usuario. Para esto podemos usar la función `input`, como se muestra en el Código 1.2.

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

```
1 def hola(nombre):
2     return "Hola " + nombre + "!"
3
4 def saludar():
5     nombre = input("Por favor ingrese su nombre: ")
6     saludo = hola(nombre)
7     print(saludo)
8
9 saludar()
```

En la función `saludar` 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 llamamos a `hola` para generar el saludo, y a `print` para mostrarlo al usuario.

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

```
$ python 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 el cuadrado de todos los números enteros del intervalo $[n1, n2)$.
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:

```
$ python cuadrados.py
Se calcularán cuadrados de números
Ingrese un número entero: 5
Ingrese otro número entero: 8
25
```

```
36
49
Es todo por ahora
```

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

```
1  def imprimir_cuadrados():
2      print("Se calcularán cuadrados de números")
3
4      n1 = int(input("Ingrese un número entero: "))
5      n2 = int(input("Ingrese otro número entero: "))
6
7      for x in range(n1, n2):
8          print(x * x)
9
10     print("Es todo por ahora")
11
12 imprimir_cuadrados()
```

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 muy distintas. Por ejemplo, los números `0`, `42` y `12345678` se almacenan como números binarios ocupando todos la misma cantidad de memoria (típicamente 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
```

¿... y `return`?

Cuando introdujimos el concepto de función dijimos que una función recibe 0 o más parámetros y devuelve un resultado. Pero en la función `saludar` que escribimos en el Código 1.2 no hay ninguna instrucción `return`... es decir, `saludar` es una función que no recibe parámetros *¡y no devuelve nada!*.

Esto es perfectamente válido: no necesitamos que `saludar` reciba parámetros porque estamos utilizando la función `input` para obtener la entrada del usuario, y no necesitamos que la función devuelva nada, porque su único cometido es *imprimir* un mensaje en la pantalla.

Sin embargo, las funciones que reciben parámetros y devuelven resultados suelen ser mucho más *reutilizables*. En la unidad 3 exploraremos un poco más este concepto.

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 `cuadrados.py`:

Instrucción	Qué sucede	Estado
`print("Se calcularán cuadrados de números")`	Se despliega el texto "Se calcularán cuadrados de números" 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
`for x in range(n1, n2):`	Se asocia el primer número de `[n1,n2)` con la variable `x` y se ejecuta el cuerpo del ciclo.	`n1` → 3 `n2` → 5 `x` → 3
`    print(x * x)`	Se imprime por pantalla el valor de `x * x` (9)	`n1` → 3 `n2` → 5 `x` → 3
`for x in range(n1, n2):`	Se asocia el segundo número de `[n1,n2)` con la variable `x` y se ejecuta el cuerpo del ciclo.	`n1` → 3 `n2` → 5 `x` → 4

`print(x*x)`	Se imprime por pantalla el valor de x * x (16)	`n1` → 3 `n2` → 5 `x` → 4
`for x in range(n1, n2):`	Como no quedan más valores por tratar en `[n1, n2)`, se sale del ciclo.	`n1` → 3 `n2` → 5 `x` → 4
`print("Es todo por ahora")`	Se despliega por pantalla el mensaje "Es todo por ahora"	`n1` → 3 `n2` → 5 `x` → 4

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:

```
def imprimir_cuadrados():
    print("Se calcularán cuadrados de números")

    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)

    for x in range(n1, n2):
        print("el valor de x es:", x)
        print(x * x)

    print("Es todo por ahora")

imprimir_cuadrados()
```

En este caso, la salida del programa será:

```
$ python cuadrados.py
Se calcularán cuadrados de números
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 x es: 5
25
el valor de x es: 6
36
el valor de x es: 7
49
Es todo por ahora
```

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

1.10 Ejercicios

Realizar los ejercicios de la Unidad 1 de la Guía de "Enunciados de ejercicios de práctica obligatoria y opcional".

Unidad 2

Programas sencillos

En esta unidad empezaremos a resolver problemas sencillos, y a programarlos en Python.

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 Realizando un programa sencillo

Al leer un artículo en una revista norteamericana 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. 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:

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

2. **Especificación.** El programa debe solicitar la distancia expresada en cantidad de millas, cantidad de pies y cantidad de pulgadas para convertirla a metros mediante la fórmula:

   $cant.metros = 1609.344 * cant.millas + 0.3048 * cant.pies + 0.0254 * cant.pulgadas$

   - Entrada:
     cant. de millas
     cant. de pies
     cant. de pulgadas
   - Salida:
     cant. de metros

   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: la solución se encuentra en el Código 2.1.

   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.** 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 ejecuciones del programa `ametrico.py`.

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

```
def main():
    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")

main()
```

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.

2.3 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.

2.3.1 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 33 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
```

2.3.2 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.
- 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 1.3, 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:

  - `input()` produce el valor ingresado por teclado tal como se lo digita.
  - `abs(x)` produce el valor absoluto del número pasado por parámetro.

Ejercicio 2.1. Aplicando las reglas matemáticas de asociatividad, decidir cuáles de las siguientes expresiones son iguales entre sí:

a) `((b * b) - (4 * a * c)) / (2 * a)`

b) `((b * b) - (4 * a * c)) // (2 * a)`

c) `(b * b - 4 * a * c) / (2 * a)`

d) `b * b - 4 * a * c / 2 * a`

e) `(b * b) - (4 * a * c / 2 * a)`

f) `1 / 2 * b`

g) `b / 2`

Ejercicio 2.2. Escribir un programa que le asigne a `a`, `b` y `c` los valores 10, 100 y 1000 respectivamente y evalúe las expresiones del ejercicio anterior.

Ejercicio 2.3. Escribir un programa que le asigne a `a`, `b` y `c` los valores 10.0, 100.0 y 1000.0 respectivamente y evalúe las expresiones del ejercicio anterior.

2.3.3 No sólo de números viven los programas

No sólo tendremos expresiones numéricas en un programa Python. Recordemos el programa que se usó para saludar a muchos amigos:

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

Para invocar a ese programa y hacer que saludara a Ana había que escribir `hola("Ana")`. La variable `alguien` en dicha invocación queda ligada a un valor que es una *cadena de caracteres* (letras, dígitos, símbolos, etc.), en este caso, `"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'
> ```

2.3.4 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 `return <expresión>`, que provoca que una función devuelva el valor resultante de evaluar la expresión.

- 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 :("
```

2.3.5 Ciclos definidos

Algunas instrucciones son compuestas, como por ejemplo la instrucción `for`, que indica a Python que queremos inicializar un *ciclo definido*:

```
for x in range(n1, n2):
    print(x * x)
```

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>`.

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.

2.4 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?

2.5 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).

2.6 Ejercicios

Realizar los ejercicios de la Unidad 2 de la Guía de "Enunciados de ejercicios de práctica obligatoria y opcional".

Unidad 3

Funciones

En la primera unidad vimos que el programador puede definir nuevas instrucciones, que llamamos *funciones*. En particular lo aplicamos a la construcción de una función llamada `hola` que saluda a todos a quienes queramos saludar:

```
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 se calcula a partir del argumento.

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

En general, las funciones pueden recibir ninguno, uno o más parámetros (separados por comas), y pueden o no devolver un resultado.

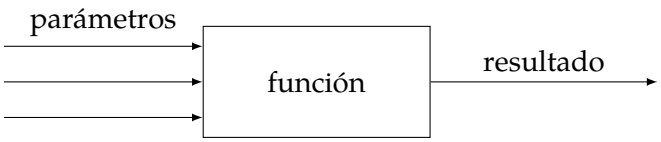


Figura 3.1: Una función recibe parámetros y devuelve un resultado.

3.1 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 se 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 2.3.4, 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`:

```
$ python -i saludo
>>> help(hola)
Help on function hola in module __main__:

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.

3.2 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`.

Ejercicio 3.1. Escribir una función `repite_hola` que reciba como parámetro un número entero `n` y escriba por pantalla el mensaje `"Hola"` `n` veces. Invocarla con distintos valores de `n`.

Ejercicio 3.2. Escribir otra función `repite_hola` que reciba como parámetro un número entero `n` y devuelva la cadena formada por `n` concatenaciones de `"Hola"`. Invocarla con distintos valores de `n`.

Ejercicio 3.3. Escribir una función `repite_saludo` que reciba como parámetro un número entero `n` y una cadena `saludo` y escriba por pantalla el valor de `saludo` `n` veces. Invocarla con distintos valores de `n` y de `saludo`.

Ejercicio 3.4. Escribir otra función `repite_saludo` que reciba como parámetro un número entero `n` y una cadena `saludo` devuelva el valor de `n` concatenaciones de `saludo`. Invocarla con distintos valores de `n` y de `saludo`.

3.3 Cómo usar una función en un programa

Las funciones son útiles porque nos permiten repetir la misma 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 las tiene que mostrar en pantalla expresadas en segundos.

   Una duración expresada en H horas, M minutos y S segundos, se convierte a segundos calculando:

   $60 * 60 * H + 60 * M + S$

2. **Especificación:** Debe solicitar tres duraciones expresadas en horas, minutos y segundos, y las tiene que mostrar en pantalla convertidas en segundos mediante la fórmula:

   $segundos = 3600 * cant.horas + 60 * cant.minutos + cant.segundos$

   - **Entradas:** Tres duraciones leídas de teclado:
     cant. de horas
     cant. de minutos
     cant. de segundos
   - **Salidas:** Mostrar por pantalla tres duraciones convertidas:
     cant. de segundos

3. **Diseño:**

   - Se tienen que leer tres conjuntos de datos y para cada conjunto hacer lo mismo; se trata entonces de un programa con estructura de ciclo definido de tres pasos:

     ```
     repetir 3 veces:
         <hacer cosas>
     ```

   - El cuerpo del ciclo (`<hacer cosas>`) tiene la estructura *Entrada-Cálculo-Salida*. En pseudocódigo:

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

     Leer cuántos minutos 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 el cuerpo del ciclo se diseñe como:

     ```
     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 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:

```
repetir 3 veces:
    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      for x in range(3):
10         h = int(input("Cuantas horas?: "))
11         m = int(input("Cuantos minutos?: "))
12         s = int(input("Cuantos segundos?: "))
13         print("Son", a_segundos(h, m, s), "segundos")
14
15 main()
```

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

```
$ python 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
```

3.4 Alcance de las variables

Ya hemos visto que podemos definir variables, ya sea dentro o fuera del cuerpo de una función. Veamos un ejemplo, utilizando la función `suma_cuadrados` de la unidad 1:

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

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

¿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)
```

1. Se invoca a `suma_cuadrados` con el argumento 5, y se ejecuta el cuerpo de la función con la variable local `n` → 5.
2. La función declara una variable local `suma` → 0.
3. Cuando la ejecución llega a la línea `return suma`, la variable `suma` → 55. Por lo tanto, la función devuelve el valor 55.
4. La función termina su ejecución, y con ella dejan de existir todas sus variables locales: `n` y `suma`.
5. Se declara la variable `y` → 55, que es el valor que devolvió la función.

Si la función no devolviera ningún valor, la variable `y` no quedaría asociada a ningún valor[1].

3.5 Un ejemplo completo

Problema 3.1. Un usuario nos plantea su problema: necesita que se facture el uso de un teléfono. Nos informará la tarifa por segundo, cuántas comunicaciones se realizaron, la duración de cada comunicación expresada en horas, minutos y segundos. Como resultado deberemos informar la duración en segundos de cada comunicación y su costo.

Solución. Aplicaremos los pasos aprendidos:

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

1. **Análisis:**
   - ¿Cuántas tarifas distintas se usan? Una sola (la llamaremos p).
   - ¿Cuántas comunicaciones se realizaron? La cantidad de comunicaciones (a la que llamaremos n) se informa cuando se inicia el programa.
   - ¿En qué formato vienen las duraciones de las comunicaciones? Vienen como ternas (h, m, s).
   - ¿Qué se hace con esas ternas? Se convierten a segundos y se calcula el costo de cada comunicación multiplicando el tiempo por la tarifa.
2. **Especificación:**
   - **Entradas:**
     - Una tarifa p expresada en pesos/segundo.
     - Una cantidad n de llamadas telefónicas.
     - n duraciones de llamadas leídas de teclado y expresadas en horas, minutos y segundos.
   - **Salidas:** Mostrar por pantalla las n duraciones ingresadas, convertidas a segundos, y su costo. Para cada juego de datos de entrada (h, m, s) se imprime:

$$3600h + 60m + s$$

$$p \cdot (3600h + 60m + s)$$

3. **Diseño:**

   Lo primero que hacemos es buscar un programa que haga algo análogo y ver si se lo puede modificar para resolver nuestro problema. Hay similitudes entre el requerimiento y el programa `tres_tiempos` que desarrollamos anteriormente. Veamos las diferencias entre sus especificaciones.

`tres_tiempos.py`	`tarifador.py`

`repetir 3 veces:`
`  <hacer cosas>`
	`leer el valor de p`
	`leer el valor de n`
	`repetir n veces:`
	`  <hacer cosas>`

El cuerpo del ciclo:
`Leer el valor de h`
`Leer el valor de m`
`Leer el valor de s`
`Mostrar a_segundos(h, m, s)`
	El cuerpo del ciclo:
	`Leer el valor de h`
	`Leer el valor de m`
	`Leer el valor de s`
	`duracion = a_segundos(h, m, s)`
	`costo = duracion * p`
	`Mostrar duracion y costo`

4. **Implementación:** El programa resultante se muestra en el Código 3.2.
5. **Prueba:** Lo probamos con una tarifa de $ 0.40 el segundo y tres ternas de `(1,0,0)`, `(0,1,0)` y `(0,0,1)`:

Código 3.2 `tarifador.py`**:** Programa para calcular el costo de uso de un teléfono.

```
def main():
    """El usuario ingresa la tarifa por segundo, cuántas
       comunicaciones se realizaron, y la duracion de cada
       comunicación expresada en horas, minutos y segundos. Como
       resultado se informa la duración en segundos de cada
       comunicación y su costo."""

    p = float(input("¿Cuánto cuesta 1 segundo de comunicacion?: "))
    n = int(input("¿Cuántas comunicaciones hubo?: "))
    for x in range(n):
        h = int(input("¿Cuántas horas?: "))
        m = int(input("¿Cuántos minutos?: "))
        s = int(input("¿Cuántos segundos?: "))
        duracion = a_segundos(h, m, s)
        costo = duracion * p
        print("Duracion:", duracion, "segundos. Costo: $", costo)

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

main()
```

```
$ python tarifador.py
Cuanto cuesta 1 segundo de comunicacion?: 0.40
Cuantas comunicaciones hubo?: 3
Cuantas horas?: 1
Cuantos minutos?: 0
Cuantos segundos?: 0
Duracion: 3600 segundos. Costo: $ 1440.0
Cuantas horas?: 0
Cuantos minutos?: 1
Cuantos segundos?: 0
Duracion: 60 segundos. Costo: $ 24.0
Cuantas horas?: 0
Cuantos minutos?: 0
Cuantos segundos?: 1
Duracion: 1 segundos. Costo: $ 0.4
```

6. **Mantenimiento:**

   Ejercicio 3.5. Corregir el programa para que:

   - Informe el costo en pesos y centavos, en lugar de un número decimal.
   - Informe cuál fue el total facturado en la corrida.

3.6 Devolver múltiples resultados

Problema 3.2. 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 3.3 `saludos.py`**:** Módulo con funciones para saludar

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

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

Código 3.4 `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 3.3 y Código 3.4 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!

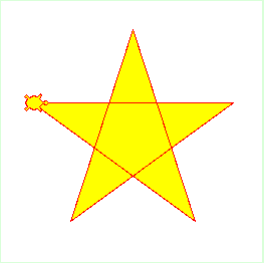
Sabías que...

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

turtle.shape("turtle")
turtle.color('red', 'yellow')
turtle.begin_fill()
for i in range(5):
    turtle.forward(200)
    turtle.right(144)
turtle.end_fill()
```



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 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
```

3.9 Ejercicios

Realizar los ejercicios de la Unidad 3 de la Guía de "Enunciados de ejercicios de práctica obligatoria y opcional".

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 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`

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

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>`.

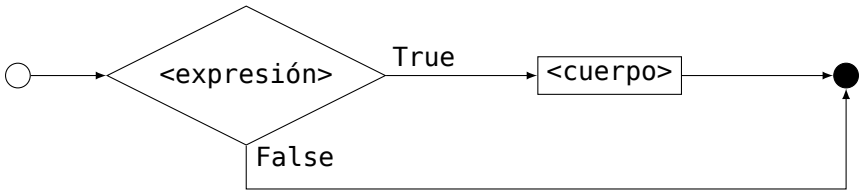


Figura 4.1: Diagrama de flujo para la instrucción `if`.

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
```

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`.

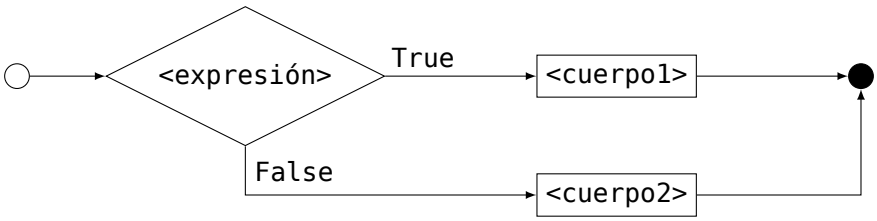


Figura 4.2: Diagrama de flujo para la estructura `if-else`.

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
```

4.3 Múltiples decisiones consecutivas

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.

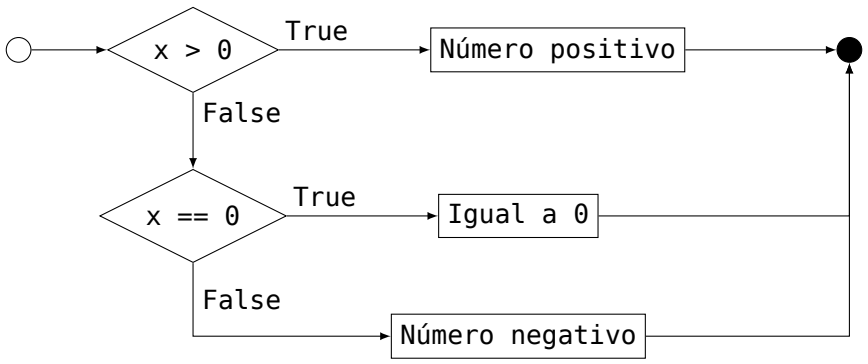


Figura 4.3: Diagrama de flujo para una estructura `if-elif-else`.

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:
```

4.4 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

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

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`

4.5 Ejercicios

Realizar los ejercicios de la Unidad 4 de la Guía de "Enunciados de ejercicios de práctica obligatoria y opcional".

Unidad 5

Más sobre ciclos

El último problema analizado en la unidad anterior decía:

> 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".

Se nos plantea a continuación un nuevo problema, similar al anterior:

Problema 5.1. 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.

5.1 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`.

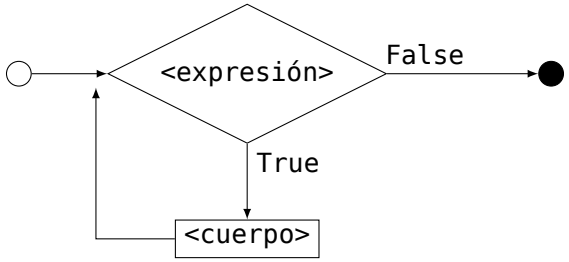


Figura 5.1: Diagrama de flujo para el ciclo indefinido `while`.

5.2 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
```

5.3 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 la 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()
```

Esta implementación es mejor que la anterior: si tuviéramos que cambiar el mensaje sólo tendríamos que modificar una línea de código. Pero tenemos que recordar inicializar la variable `centinela` antes de comenzar el ciclo.

5.4 Ejercicios

Ejercicio 5.1. Nuevamente, se desea facturar el uso de un teléfono. Para ello se informa la tarifa por segundo y la duración de cada comunicación expresada en horas, minutos y segundos. Como resultado se informa la duración en segundos de cada llamado y su costo. Resolver este problema usando

1. Un ciclo definido.
2. Un ciclo interactivo.
3. Un ciclo con centinela.

Ejercicio 5.2. Mantenimiento del tarifador: al final del día se debe informar cuántas llamadas hubo y el total facturado. Hacerlo con todos los esquemas anteriores.

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.
- Además de la condición que hace que el ciclo se termine, es posible interrumpir su ejecución con código específico dentro del ciclo.

Referencia Python

`while <condicion>:`

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

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

`break`

Interrumpe la ejecución del ciclo actual. Puede utilizarse tanto para ciclos definidos como indefinidos.

```
return [valor]
```

Finaliza la ejecución de una función, y además corta la ejecución del ciclo actual, en caso de estar dentro del cuerpo del ciclo.

5.6 Ejercicios

Realizar los ejercicios de la Unidad 5 de la Guía de "Enunciados de ejercicios de práctica obligatoria y opcional".

Unidad 6

Cadenas de caracteres

Una cadena es una secuencia de caracteres. Ya las hemos usado 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, 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(12)
<class 'int'>
>>> type(12.0)
<class 'float'>
>>> type(True)
<class 'bool'>
>>> type("Hola")
<class 'str'>
```

6.1 Operaciones con cadenas

Ya vimos en la sección 2.3.3 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 '
```

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

A continuación, otras operaciones y particularidades de las cadenas.

6.1.1 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 apóstrofo o comilla que abre, y un apóstrofo o comilla que cierra), y que por lo tanto tiene longitud cero:

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

6.1.2 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

>>>
```

6.1.3 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")
```

6.1.4 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
```

Ejercicio 6.1. Escribir un ciclo que permita mostrar los caracteres de una cadena del final al principio.

6.2 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]
''
```

Ejercicio 6.2. Investigar cuál es el resultado de `s[:]`.

Ejercicio 6.3. Investigar cuál es el resultado de `s[:j]` y `s[j:]` si `j` es un número negativo.

6.3 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'
```

6.4 Procesamiento sencillo de cadenas

Problema 6.1. 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 que representa cuántas letras "A" tiene `s`.

   - Entrada: cadena de caracteres
   - Salida: cantidad de letras

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
   ```

   Se necesita una variable `contador` que cuente 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? Alicia Hacker nos hace notar que 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? E. Lapurado, nuestro alumno impaciente nos induce a poner manos a la obra y a programar esta solución, y el resto del curso está de acuerdo.

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`? `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)**

```
def contarA(s):
    """Devuelve cuántas letras "A" aparecen en la cadena s."""
    contador = 0
    for letra in s:
        if letra == "A":
            contador = contador + 1
    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. **Mantenimiento:**

   Esta función resulta un poco limitada. Cuando necesitemos contar cuántas letras "E" hay en una cadena tendremos que hacer otra función. Tiene sentido hacer una función más general que nos permita contar cuántas veces aparece un carácter dado en una cadena.

   Ejercicio 6.4. Escribir una función `contar(c, s)` que cuente cuántas veces aparece un carácter `c` dado en una cadena `s`.

6.5 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.

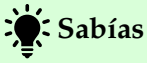
Por ejemplo, si queremos saludar al usuario y mostrarle la cantidad de mensajes sin leer, y en el estado tenemos `nombre` → `'Veronika'` y `no_leidos` → `8`, podemos hacer algo como:

```
>>> "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`.

Esta notación es propia de la *orientación a objetos*.

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 `{:.2f}` para indicar dos dígitos luego del separador decimal.

```
>>> precio = 205.5
>>> 'Sin IVA: ${:.2f}. Con IVA: ${:.2f}'.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:.2f}. Con IVA: ${precio * 1.21:.2f}'
'Sin IVA: $205.50. Con IVA: $248.66'
>>> f'Glóbulos rojos: {rojos:.1e}/uL'
'Glóbulos rojos: 4.6e+06/uL'
```

6.6 Nuestro primer juego

Con todo esto ya estamos en condiciones de escribir un programa para jugar con la computadora: el *Mastermind*. El Mastermind es un juego que consiste en deducir un código numérico de (por ejemplo) cuatro cifras.

1. **Análisis** (explicación del juego):

   Cada vez que se empieza un partido, el programa debe "elegir" un número de cuatro cifras (sin cifras repetidas), que será el código que el jugador debe adivinar en la menor cantidad de intentos posibles. Cada intento consiste en una propuesta de un código posible que tipea el jugador, y una respuesta del programa. Las respuestas le darán pistas al jugador para que pueda deducir el código.

   Estas pistas indican cuán cerca estuvo el número propuesto de la solución a través de dos valores: la cantidad de *aciertos* es la cantidad de dígitos que propuso el jugador que también están en el código *en la misma posición*. La cantidad de *coincidencias* es la cantidad de digitos que propuso el jugador que también están en el código pero *en una posición distinta*.

   Por ejemplo, si el código que eligió el programa es el `2607`, y el jugador propone el `1406`, el programa le debe responder un acierto (el `0`, que está en el código original en el mismo lugar, el tercero), y una coincidencia (el `6`, que también está en el código original, pero en la segunda posición, no en el cuarto como fue propuesto). Si el jugador hubiera propuesto el `3591`, habría obtenido como respuesta ningún acierto y ninguna coincidencia, ya que no hay números en común con el código original, y si se obtienen cuatro aciertos es porque el jugador adivinó el código y ganó el juego.

2. **Especificación:** El programa, entonces, debe generar un número que el jugador no pueda predecir. A continuación, debe pedirle al usuario que introduzca un número de cuatro cifras distintas, y cuando éste lo ingresa, procesar la propuesta y evaluar el número de aciertos y de coincidencias que tiene de acuerdo al código elegido. Si es el código original, se termina el programa con un mensaje de felicitación. En caso contrario, se informa al jugador la cantidad de aciertos y la de coincidencias, y se le pide una nueva propuesta. Este proceso se repite hasta que el jugador adivine el código.

3. **Diseño:**

Lo primero que tenemos que hacer es indicarle al programa que tiene que "elegir" un número de cuatro cifras al azar. Esto lo hacemos a través del módulo `random`. Este módulo provee funciones para hacer elecciones aleatorias[1].

La función del módulo que vamos a usar se llama `choice`. Esta función recibe una secuencia de valores, y devuelve un valor de la secuencia elegido al azar. Una n-upla (ver sección 3.6) es un ejemplo de una secuencia, por lo que podemos hacer algo como:

```
>>> import random
>>> digitos = ('0','1','2','3','4','5','6','7','8','9')
>>> random.choice(digitos)
'4'
>>> random.choice(digitos)
'1'
```

Como están entre comillas, los dígitos son tratados como cadenas de caracteres de longitud uno. Sin las comillas, habrían sido considerados números enteros. En este caso elegimos verlos como cadenas de caracteres porque lo que nos interesa hacer con ellos no son cuentas sino comparaciones, concatenaciones, contar cuántas veces aparece o donde está en una cadena de mayor longitud, es decir, las operaciones que se aplican a cadenas de texto. Entonces que sean variables de tipo cadena de caracteres es lo que mejor se adapta a nuestro problema.

Ahora tenemos que generar el número al azar, asegurándonos de que no haya cifras repetidas. Esto lo podemos modelar así:

```
Tomar una cadena vacía
Repetir cuatro veces:
    1. Elegir un elemento al azar de la lista de dígitos
    2. Si el elemento no está en la cadena, agregarlo
    3. En caso contrario, volver al punto 1
```

Una vez elegido el número, hay que interactuar con el usuario y pedirle su primera propuesta. Si el número no coincide con el código, hay que buscar la cantidad de aciertos y de coincidencias y repetir el pedido de propuestas, hasta que el jugador adivine el código.

Para verificar la cantidad de aciertos se pueden recorrer las cuatro posiciones de la propuesta: si alguna coincide con los dígitos en el código en esa posición, se incrementa en uno la cantidad de aciertos. En caso contrario, se verifica si el dígito está en alguna otra posición del código, y en ese caso se incrementa la cantidad de coincidencias. En cualquier caso, hay que incrementar en uno también la cantidad de intentos que lleva el jugador.

Finalmente, cuando el jugador acierta el código elegido, hay que dejar de pedir propuestas, informar al usuario que ha ganado y terminar el programa.

4. **Implementación:** Entonces, de acuerdo a lo diseñado en 3, se muestra una implementación en el Código 6.1.

[1] En realidad, la computadora nunca puede hacer elecciones *completamente* aleatorias. Por eso los números "al azar" que puede elegir se llaman *pseudoaleatorios*.

Código 6.1 `mastermind.py`: Juego Mastermind

```
import random

def mastermind():
    """Funcion principal del juego Mastermind"""
    print("Bienvenid@ al Mastermind!")
    print("Tienes que adivinar un numero de cuatro cifras distintas")

    codigo = elegir_codigo()
    intentos = 1
    propuesta = input("Que codigo propones?: ")

    while propuesta != codigo:
        intentos += 1
        aciertos, coincidencias = analizar_propuesta(propuesta, codigo)
        print("Tu propuesta ({}) tiene {} aciertos y {} coincidencias.".format(
            propuesta,
            aciertos,
            coincidencias
        ))
        propuesta = input("Propone otro codigo: ")

    print("Felicitaciones! Adivinaste el codigo en {} intentos.".format(intentos))

def elegir_codigo():
    """Devuelve un codigo de 4 digitos elegido al azar"""
    digitos = ('0','1','2','3','4','5','6','7','8','9')
    codigo = ''
    for i in range(4):
        candidato = random.choice(digitos)
        # Debemos asegurarnos de no repetir digitos
        while candidato in codigo:
            candidato = random.choice(digitos)
        codigo = codigo + candidato
    return codigo

def analizar_propuesta(propuesta, codigo):
    """Determina la cantidad de aciertos y coincidencias"""
    aciertos = 0
    coincidencias = 0
    for i in range(4):
        if propuesta[i] == codigo[i]:
            aciertos = aciertos + 1
        elif propuesta[i] in codigo:
            coincidencias = coincidencias + 1
    return aciertos, coincidencias

mastermind()
```

5. **Pruebas:** La forma más directa de probar el programa es jugándolo, y verificando manualmente que las respuestas que da son correctas, por ejemplo:

```
$ python mastermind.py
Bienvenido/a al Mastermind!
Tenes que adivinar un numero de 4 cifras distintas
Que codigo propones?: 1234
Tu propuesta (1234) tiene 0 aciertos y  1 coincidencias.
Propone otro codigo: 5678
Tu propuesta (5678) tiene 0 aciertos y  1 coincidencias.
Propone otro codigo: 1590
Tu propuesta (1590) tiene 1 aciertos y  1 coincidencias.
Propone otro codigo: 2960
Tu propuesta (2960) tiene 2 aciertos y  1 coincidencias.
Propone otro codigo: 0963
Tu propuesta (0963) tiene 1 aciertos y  2 coincidencias.
Propone otro codigo: 9460
Tu propuesta (9460) tiene 1 aciertos y  3 coincidencias.
Propone otro codigo: 6940
Felicitaciones! Adivinaste el codigo en 7 intentos.
```

Podemos ver que para este caso el programa parece haberse comportado bien. ¿Pero cómo podemos saber que el código final era realmente el que eligió originalmente el programa? ¿O qué habría pasado si no encontrábamos la solución?

Para probar estas cosas recurrimos a la depuración del programa. Una forma de hacerlo es simplemente agregar algunas líneas en el código que nos informen lo que está sucediendo que no podemos ver. Por ejemplo, los números que va eligiendo al azar y el código que queda al final. Así podremos verificar si las respuestas son correctas a medida que las hacemos y podremos elegir mejor las propuestas en las pruebas.

```
def elegir_codigo():
    """Devuelve un codigo de 4 digitos elegido al azar"""
    digitos = ('0','1','2','3','4','5','6','7','8','9')
    codigo = ''
    for i in range(4):
        candidato = random.choice(digitos)
        print('[DEBUG] candidato:', candidato)
        # Debemos asegurarnos de no repetir digitos
        while candidato in codigo:
            candidato = random.choice(digitos)
            print('[DEBUG] otro candidato:', candidato)
        codigo = codigo + candidato
        print('[DEBUG] el codigo va siendo:', codigo)
    return codigo
```

De esta manera podemos monitorear cómo se va formando el código que hay que adivinar, y los candidatos que van apareciendo pero se rechazan por estar repetidos:

```
$ python master_debug.py
Bienvenido/a al Mastermind!
Tienes que adivinar un numero de cuatro cifras distintas
[DEBUG] candidato: 8
[DEBUG] el codigo va siendo: 8
[DEBUG] candidato: 0
```

```
[DEBUG] el codigo va siendo: 80
[DEBUG] candidato: 2
[DEBUG] el codigo va siendo: 802
[DEBUG] candidato: 8
[DEBUG] otro candidato: 2
[DEBUG] otro candidato: 7
[DEBUG] el codigo va siendo: 8027
Que codigo propones?:
```

6. **Mantenimiento:** Supongamos que queremos jugar el mismo juego, pero en lugar de hacerlo con un número de cuatro cifras, adivinar uno de cinco. ¿Qué tendríamos que hacer para cambiarlo?

   Para empezar, habría que reemplazar el 4 en la línea 28 del programa por un 5, indicando que hay que elegir 5 dígitos al azar. Pero además, el ciclo en la línea 40 también necesita cambiar la cantidad de veces que se va a ejecutar, 5 en lugar de 4. Y hay un lugar más, adentro del mensaje al usuario que indica las instrucciones del juego en la línea 6.

   El problema de ir cambiando estos números de a uno es que si quisiéramos volver al programa de los 4 dígitos o quisiéramos cambiarlo por uno que juegue con 3, tenemos que volver a hacer los reemplazos en todos lados cada vez que lo queremos cambiar, y corremos el riesgo de olvidarnos de alguno e introducir errores en el código.

   Una forma de evitar esto es fijar la cantidad de cifras en una variable:

```
import random

CANT_DIGITOS = 5

def mastermind():
    """Funcion principal del juego Mastermind"""
    print("Bienvenido/a al Mastermind!")
    print("Tienes que adivinar un numero de {} cifras distintas".format(
    ↪ CANT_DIGITOS))
```

   Por convención, el nombre de la variable se escribe en mayúsculas, para indicar que el valor asignado es constante a lo largo de la ejecución del programa.

   En la función `elegir_codigo`:

```
def elegir_codigo():
    """Devuelve un codigo de CANT_DIGITOS digitos elegido al azar"""
    digitos = ('0','1','2','3','4','5','6','7','8','9')
    codigo = ''
    for i in range(CANT_DIGITOS):
```

   Y el chequeo de aciertos y coincidencias:

```
def analizar_propuesta(propuesta, codigo):
    """Determina la cantidad de aciertos y coincidencias"""
    aciertos = 0
    coincidencias = 0
    for i in range(CANT_DIGITOS):
```

   Con 5 dígitos, el juego se pone más difícil. Nos damos cuenta que si el jugador no logra adivinar el código, el programa no termina: se queda preguntando códigos y respondiendo aciertos y coincidencias para siempre. Entonces queremos darle al usuario la posibilidad de rendirse y saber cuál era la respuesta y terminar el programa.

Para esto agregamos en el ciclo `while` principal una condición extra: para seguir preguntando, la propuesta tiene que ser distinta al código pero además tiene que ser distinta del texto `"Me doy"`.

```
def mastermind():
    ...
    propuesta = input("Que codigo propones?: ")
    ME_DOY = "Me doy"

    while propuesta != codigo and propuesta != ME_DOY:
```

Entonces, ahora no sólamente sale del `while` si acierta el código, sino además si se rinde y quiere saber cuál era el código. Entonces afuera del while tenemos que separar las dos posibilidades, y dar distintos mensajes:

```
    if propuesta == ME_DOY:
        print("Mala suerte! El código era: {}".format(codigo))
    else:
        print("Felicitaciones! Adivinaste el codigo en {} intentos.".format(
    ↪ intentos))
```

En el Código 6.2 se muestra el código completo luego de aplicar las mejoras.

Código 6.2 `mastermind.py`: Juego Mastermind de 5 dígitos y con posibilidad de rendirse

```
import random

CANT_DIGITOS = 5

def mastermind():
    """Funcion principal del juego Mastermind"""
    print("Bienvenido/a al Mastermind!")
    print("Tienes que adivinar un numero de {} cifras distintas".format(
    ↪ CANT_DIGITOS))

    codigo = elegir_codigo()
    intentos = 1
    propuesta = input("Que codigo propones?: ")
    ME_DOY = "Me doy"

    while propuesta != codigo and propuesta != ME_DOY:
        intentos += 1
        aciertos, coincidencias = analizar_propuesta(propuesta, codigo)
        print("Tu propuesta ({}) tiene {} aciertos y {} coincidencias.".format(
            propuesta,
            aciertos,
            coincidencias
        ))
        propuesta = input("Propone otro codigo: ")

    if propuesta == ME_DOY:
        print("Mala suerte! El código era: {}".format(codigo))
    else:
        print("Felicitaciones! Adivinaste el codigo en {} intentos.".format(
    ↪ intentos))

def elegir_codigo():
    """Devuelve un codigo de CANT_DIGITOS digitos elegido al azar"""
    digitos = ('0','1','2','3','4','5','6','7','8','9')
    codigo = ''
    for i in range(CANT_DIGITOS):
        candidato = random.choice(digitos)
        # Debemos asegurarnos de no repetir digitos
        while candidato in codigo:
            candidato = random.choice(digitos)
        codigo = codigo + candidato
    return codigo

def analizar_propuesta(propuesta, codigo):
    """Determina la cantidad de aciertos y coincidencias"""
    aciertos = 0
    coincidencias = 0
    for i in range(CANT_DIGITOS):
        if propuesta[i] == codigo[i]:
            aciertos = aciertos + 1
        elif propuesta[i] in codigo:
            coincidencias = coincidencias + 1
    return aciertos, coincidencias

mastermind()
```

6.7 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 uno de los caracteres 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.format(valores)`

Genera otra cadena reemplazando las marcas de formato por el valor correspondiente, luego de convertirlo de acuerdo a lo especificado en la marca de formato.
Las marcas de formato se indican con `{}`, incluyendo opcionalmente diversos parámetros de conversión.
A continuación una referencia de algunos tipos de conversión utilizados frecuentemente:

Tipo	Significado	Ejemplo
`{:d}`	Valor entero en decimal	`'{:d}'.format(10.5)` → `'10'`
`{:o}`	Valor entero en octal	`'{:o}'.format(8)` → `'10'`
`{:x}`	Valor entero en hexadecimal	`'{:x}'.format(16)` → `'10'`
`{:f}`	Valor de punto flotante, en decimal	`'{:f}'.format(0.1**5)` → `'0.000010'`
`{:.2f}`	Punto flotante, con dos dígitos de precisión	`'{:.2f}'.format(0.1**5)` → `'0.00'`
`{:e}`	Punto flotante, en notación exponencial	`'{:e}'.format(0.1**5)` → `'1.000000e-05'`
`{:g}`	Punto flotante, lo que sea más corto	`'{:g}'.format(0.1**5)` → `'0.00001'`
`{:.2s}`	Cadena recortada en 2 caracteres	`'{:.2s}'.format('Python')` → `'Py'`
`{:<6s}`	Cadena alineada a la izquierda, ocupando 6 caracteres	`'{:<6s}'.format('Py')` → `'Py    '`
`{:^6s}`	Cadena centrada	`'{:^6s}'.format('Py')` → `'  Py  '`
`{:>6s}`	Cadena alineada a la derecha	`'{:>6s}'.format('Py')` → `'    Py'`
`{{`	Un {	
`}}`	Un }	

`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 una copia de la cadena, con solamente su primer carácter en mayúscula.

`cadena.upper()`

Devuelve una copia de la cadena convertida a mayúsculas.

`cadena.lower()`

Devuelve una copia de la cadena convertida a minúsculas.

6.8 Ejercicios

Realizar los ejercicios de la Unidad 6 de la Guía de "Enunciados de ejercicios de práctica obligatoria y opcional".

Unidad 7

Tuplas y listas

Python cuenta con una gran variedad de tipos de datos que permiten representar la información según cómo esté estructurada. En esta unidad se estudian las tuplas y las listas, que son tipos de datos utilizados cuando se quiere agrupar elementos.

7.1 Tuplas

Al diseñar algoritmos, es muy común que queramos describir un agrupamiento de datos de distintos tipos.

Esto es algo que ya hicimos anteriormente: en la conversión de un tiempo a horas, minutos y segundos (sección 3.6) y también en el juego Mastermind (sección 6.6), usamos n-uplas, que en Python se llaman *tuplas*, y sirven para representar agrupaciones de datos ordenados.

Veamos más ejemplos:

- Una fecha la podemos querer representar como la terna día (un número entero), mes (una cadena de caracteres), y año (un número entero), y tendremos por ejemplo la tupla `(25, "Mayo", 1810)`.
- Como datos de los alumnos queremos guardar número de padrón, nombre y apellido, como por ejemplo `(89766, "Alicia", "Hacker")`.
- **Es posible anidar tuplas:** como datos de los alumnos queremos guardar número de padrón, nombre, apellido y fecha de nacimiento, como por ejemplo:
  `(89766, "Alicia", "Hacker", (9, "Julio", 1988))`.

En Python el tipo de dato asociado a las tuplas se llama `tuple`:

```
>>> fecha = (25, "Mayo", 1810)
>>> type(fecha)
<class 'tuple'>
```

7.1.1 Elementos y segmentos de tuplas

Las tuplas son *secuencias*, igual que las cadenas, y se puede utilizar la misma notación de índices que en las cadenas para obtener cada una de sus componentes:

```
>>> fecha = (25, "Mayo", 1810)
>>> fecha[0]
25
>>> fecha[1]
'Mayo'
>>> fecha[2]
1810
```

Atención

Todas las secuencias en Python comienzan a numerarse desde 0. Es por eso que se produce un error si se quiere acceder al n-ésimo elemento de un tupla:

```
>>> fecha[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: tuple index out of range
```

También se puede utilizar la notación de rangos, que se vio aplicada a cadenas, para obtener una nueva tupla, con un subconjunto de componentes. Si en el ejemplo de la fecha queremos quedarnos con un par que sólo contenga día y mes podremos tomar el rango `[:2]` de la misma:

```
>>> fecha[:2]
(25, 'Mayo')
```

Ejercicio 7.1. ¿Cuál es el resultado de obtener el cuarto elemento de la tupla `(89766, "Alicia", "Hacker", (9, "Julio", 1988))`?

7.1.2 Las tuplas son inmutables

Al igual que con las cadenas, las componentes de las tuplas no pueden ser modificadas:

```
>>> fecha[2] = 2018
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

7.1.3 Longitud de tuplas

A las tuplas también se les puede aplicar la función `len()` para calcular su longitud. El valor de esta función aplicada a una tupla nos indica cuántas componentes tiene esa tupla.

```
>>> len(fecha)
3
```

Ejercicio 7.2. ¿Cuál es la longitud de la tupla `(89766, "Alicia", "Hacker", (9, "Julio", 1988))`?

- Una *tupla vacía* es una tupla con 0 componentes, y se la indica como `()`.

```
>>> z = ()
>>> len(z)
0
```

```
>>> z[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: tuple index out of range
```

- Una *tupla unitaria* es una tupla con una componente. Para distinguir la tupla unitaria de la componente que contiene, la sintaxis de Python exige que a la componente no sólo se la encierre entre paréntesis sino que se le ponga una coma a continuación del valor de la componente (así, `(1810)` es un número, pero `(1810,)` es la tupla unitaria cuya única componente vale 1810).

```
>>> u = (1810)
>>> len(u)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: object of type 'int' has no len()
>>> u = (1810,)
>>> len(u)
1
>>> u[0]
1810
```

7.1.4 Empaquetado y desempaquetado de tuplas

Si a una variable se le asigna una secuencia de valores separados por comas, el valor de esa variable será la tupla formada por todos los valores asignados. A esta operación se la denomina *empaquetado de tuplas*.

```
>>> a = 125
>>> b = "#"
>>> c = "Ana"
>>> d = a, b, c
>>> len(d)
3
>>> d
(125, '#', 'Ana')
```

Si se tiene una tupla de longitud `k`, se puede asignar la tupla a `k` variables distintas y en cada variable quedará una de las componentes de la tupla. A esta operación se la denomina *desempaquetado de tuplas*.

```
>>> x, y, z = d
>>> x
125
>>> y
'#'
>>> z
'Ana'
```

Atención

Si las variables no son distintas, se pierden valores. Y si las variables no son exactamente k se produce un error.

```
>>> p, p, p = d
>>> p
'Ana'
>>> m, n = d
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: too many values to unpack
>>> m, n, o, p = d
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: need more than 3 values to unpack
```

Sabías que...

En la implementación de algunos programas suele ser necesario intercambiar el valor de dos variables. Si queremos intercambiar el valor de `a` y `b`, una posibilidad es utilizar una variable auxiliar:

```
aux = a
a = b
b = aux
```

Pero un "truco" que permiten algunos lenguajes, entre ellos Python, es la posibilidad de empaquetar y desempaquetar una tupla en una única operación:

```
a, b = b, a
```

7.1.5 Ejercicios con tuplas

Realizar los ejercicios con tuplas, de la Unidad 7 de la Guía de "Enunciados de ejercicios de práctica obligatoria y opcional".

7.2 Listas

Presentaremos ahora una nueva estructura de datos: la *lista*. 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.

La notación para lista es una secuencia de valores encerrados entre corchetes y separados por comas. Por ejemplo, si representamos a los alumnos mediante su número de padrón, 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'>
```

7.2.1 Longitud de la lista. Elementos y segmentos de listas

- Como a las secuencias 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 de cadenas y tuplas, con valores que van de 0 a la longitud de la lista -1.

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

- Para obtener una sublista a partir de la lista original, se utiliza la notación de rangos, como en las otras secuencias.

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

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

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

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

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

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

7.2.2 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 una componente de una lista, se selecciona la componente mediante su índice y se le asigna el nuevo valor:

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

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

```
>>> padrones.append(47890)
>>> padrones
[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 `padrones.insert(2, 54988)` para insertar el padrón 54988 en la tercera posición de `padrones`.

```
>>> padrones.insert(2, 54988)
>>> padrones
[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.

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

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

  Escribimos `padrones.remove(45750)` para borrar el padrón 45750 de la lista de inscriptos:

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

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

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

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

7.2.3 Cómo buscar dentro de las listas

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

- ¿Está la persona cuyo padrón es v inscripta en esta materia?
- ¿En qué orden se inscribió la persona cuyo padrón 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`:

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

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

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

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

Atención

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

```
>>> padrones.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 y tuplas.

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

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

El ciclo `for <variable> in <secuencia>:` se puede utilizar sobre cualquier secuencia, incluyendo tuplas y 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(padrones):
...     print(i, p)
...
0 78455
1 79211
2 54988
3 66540
4 47890
```

Sabías que...

En Python, las listas, las tuplas 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 o una tupla a partir de cualquier otra secuencia, utilizando las funciones `list` y `tuple`, respectivamente:

```
>>> list("Hola")
['H', 'o', 'l', 'a']
>>> tuple("Hola")
('H', 'o', 'l', 'a')
>>> list((1, 2, 3, 4))
[1, 2, 3, 4]
```

Problema 7.1. Queremos escribir un programa que nos permita armar la lista de los inscriptos de una materia.

1. **Análisis:** El usuario ingresa datos de padrones que se van guardando en una lista.
2. **Especificación:** El programa solicitará al usuario que ingrese uno a uno los padrones 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 padrones 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 5.3, 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:

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

  Como sabemos que los números de padrón 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
  Pedir padrón
  Mientras el padrón sea positivo:
      Agregar el padrón a la lista
      Pedir padrón
  Devolver la lista de inscriptos
  ```

4. **Implementación:** De acuerdo a lo diseñado, el programa quedaría como se muestra en el Código 7.1.

Código 7.1 `inscripcion.py`**:** Permite ingresar padrones de alumnos inscriptos

```
1  def inscribir_alumnos():
2      """Permite inscribir alumnos al curso"""
3
4      print("Inscripcion en el curso de Algoritmos y Programación I")
5      inscriptos = []
6      padron = int(input("Ingresa un padrón (<=0 para terminar): "))
7      while padron > 0:
8          inscriptos.append(padron)
9          padron = int(input("Ingresa un padrón (<=0 para terminar): "))
10     return inscriptos
11
12 inscriptos = inscribir_alumnos()
13 print("La lista de inscriptos es:", inscriptos)
```

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

   ```
   $ python inscripcion.py
   Inscripcion en el curso de Algoritmos y Programación I
   Ingresa un padrón (<=0 para terminar): 30
   ```

```
Ingresa un padrón (<=0 para terminar): 40
Ingresa un padrón (<=0 para terminar): 50
Ingresa un padrón (<=0 para terminar): 0
La lista de inscriptos es: [30, 40, 50]

$ python inscripcion.py
Inscripcion en el curso de Algoritmos y Programación I
Ingresa un padrón (<=0 para terminar): 0
La lista de inscriptos es: []

$ python inscripcion.py
Inscripcion en el curso de Algoritmos y Programación I
Ingresa un padrón (<=0 para terminar): 30
Ingresa un padrón (<=0 para terminar): 40
Ingresa un padrón (<=0 para terminar): 40
Ingresa un padrón (<=0 para terminar): 30
Ingresa un padrón (<=0 para terminar): 50
Ingresa un padrón (<=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 padrones repetidos.

7. **Diseño revisado:** Para no permitir que haya padrones repetidos debemos revisar que no exista el padrón antes de agregarlo en la lista:

```
La lista de inscriptos es vacía
Pedir padrón
Mientras el padrón sea positivo:
    Si el padrón está en la lista:
        Avisar que el padrón ya está en la lista
    Si  no:
        Agregar el padrón a la lista
    Pedir padrón
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 7.2.

Código 7.2 `inscripcion.py`**:** Permite ingresar padrones, sin repetir

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

    print("Inscripcion en el curso de Algoritmos y Programación I")
    inscriptos = []
    padron = int(input("Ingresa un padrón (<=0 para terminar): "))
    while padron > 0:
        if padron in inscriptos:
            print("El padrón ya está en la lista de inscriptos.")
        else:
            inscriptos.append(padron)
        padron = int(input("Ingresa un padrón (<=0 para terminar): "))
    return inscriptos

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

9. **Nueva prueba:** Para probarlo lo ejecutamos con los mismos lotes de prueba anteriores (inscripción de tres alumnos, inscripción de cero alumnos, inscripción de alumnos repetidos):

```
$ python inscripcion.py
Inscripcion en el curso de Algoritmos y Programación I
Ingresa un padrón (<=0 para terminar): 30
Ingresa un padrón (<=0 para terminar): 40
Ingresa un padrón (<=0 para terminar): 50
Ingresa un padrón (<=0 para terminar): 0
La lista de inscriptos es: [30, 40, 50]

$ python inscripcion.py
Inscripcion en el curso de Algoritmos y Programación I
Ingresa un padrón (<=0 para terminar): 0
La lista de inscriptos es: []

$ python inscripcion.py
Inscripcion en el curso de Algoritmos y Programación I
Ingresa un padrón (<=0 para terminar): 30
Ingresa un padrón (<=0 para terminar): 40
Ingresa un padrón (<=0 para terminar): 40
El padrón ya está en la lista de inscriptos.
Ingresa un padrón (<=0 para terminar): 30
El padrón ya está en la lista de inscriptos.
Ingresa un padrón (<=0 para terminar): 50
Ingresa un padrón (<=0 para terminar): 0
La lista de inscriptos es: [30, 40, 50]
```

Ahora el resultado es satisfactorio: no tenemos inscriptos repetidos.

7.3 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 padrones 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]
```

7.4 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 "cadena con espacios" escribiremos simplemente "cadena con espacios".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'
```

7.4.1 Ejercicios con listas y cadenas

Ejercicio 7.3. Escribir una función que reciba como parámetro una cadena de palabras separadas por espacios y devuelva, como resultado, cuántas palabras de más de cinco letras tiene la cadena dada.

7.5 Resumen

- Python nos provee con varias estructuras que nos permiten agrupar los datos que tenemos. En particular, las **tuplas** son estructuras **inmutables** que permiten agrupar valores al momento de crearlas, y las **listas** son estructuras **mutables** que permiten agrupar valores, con la posibilidad de agregar, quitar o reemplazar sus elementos.
- Las tuplas se utilizan para modelar situaciones en las cuales al momento de crearlas ya se sabe cuál va a ser la información a almacenar. Por ejemplo, para representar una fecha, una carta de la baraja, una ficha de dominó.
- 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 las notas de un alumno en diversas materias, los inscriptos para un evento o la clasificación de los equipos en una competencia.
- Las cadenas, tuplas y listas son tres 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.

Referencia Python

`(valor1, valor2, valor3)`

Las tuplas se definen como una sucesión de valores encerrados entre paréntesis y separados por comas. Una vez definidas, no se pueden modificar los valores asignados.

Casos particulares:

```
tupla_vacia = ()
tupla_unitaria = (3459,)
```

`[valor1, valor2, valor3]`

Las listas se definen como una sucesión de valores encerrados entre corchetes y separados por comas. Se les puede agregar 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, mientras 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 caso de que se omita `i`, se asume `0`. En caso de que se omita `j`, se asume `len(secuencia)`. En 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 el elemento "valor" al final de la lista.

```
lista.insert(posicion, valor)
```

Agrega el elemento "valor" a la lista, en la posición `posicion`.

```
lista.remove(valor)
```

Quita de la lista la primera aparición de valor, 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.

7.6 Ejercicios

Realizar los ejercicios de la Unidad 7 de la Guía de "Enunciados de ejercicios de práctica obligatoria y opcional".

Unidad 8

Diccionarios

En esta unidad analizaremos otra estructura de datos importante: los diccionarios. Su importancia radica no sólo en las grandes posibilidades que presentan como estructuras para almacenar información, sino también en que, en Python, son utilizados por el propio lenguaje para realizar diversas operaciones y para almacenar información de otras estructuras.

8.1 Qué es un diccionario

Según Wikipedia, "[u]n diccionario es una obra de consulta de palabras y/o términos que se encuentran generalmente ordenados alfabéticamente. De dicha compilación de palabras o términos se proporciona su significado, etimología, ortografía y, en el caso de ciertas lenguas fija su pronunciación y separación silábica."

Al igual que los diccionarios a los que se refiere Wikipedia, los diccionarios de Python son una colección de términos (llamados *claves*) asociados a un *valor* determinado. A diferencia de los diccionarios a los que se refiere Wikipedia, el orden en los diccionarios de Python no es relevante.

Dicho de otra manera, un diccionario es una colección de pares (*clave*, *valor*). A diferencia de las listas y tuplas, en lugar de acceder a un valor mediante un índice numérico, el acceso será a través de su clave, que puede ser de diversos tipos.

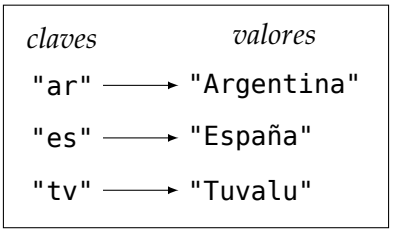


Figura 8.1: Un diccionario cuyas claves son dominios de Internet (*top level domains*) y cuyos valores son los países correspondientes.

Las claves son únicas dentro de un diccionario, es decir que no puede haber un diccionario que tenga dos veces la misma clave. Si se asigna un valor a una clave ya existente, se reemplaza el valor anterior.

No hay una forma directa de acceder a una clave a través de su valor, y nada impide que un mismo valor se encuentre asignado a distintas claves.

Dado que el orden en los diccionarios no es relevante, dos diccionarios se consideran iguales si contienen las mismas claves asociadas a los mismos valores, incluso aunque los elementos hayan sido agregados en diferente orden.

Al igual que las listas, los diccionarios son mutables. Esto significa que podemos agregar, quitar y modificar los elementos de un diccionario posteriormente a su creación.

Cualquier valor de tipo inmutable puede ser clave de un diccionario: cadenas, enteros, tuplas (con valores inmutables en sus miembros), etc. No hay restricciones para los valores que el diccionario puede contener, cualquier tipo puede ser el valor: listas, cadenas, tuplas, otros diccionarios, etc.

Sabías que...

En otros lenguajes de programación, a los diccionarios se los llama *arreglos asociativos*, *mapas* o *tablas*.

8.2 Utilizando diccionarios en Python

De la misma forma que con listas, es posible definir un diccionario directamente con los miembros que va a contener, o bien inicializar el diccionario vacío y luego agregar los valores de a uno o de a muchos.

Para definirlo junto con los miembros que va a contener, se encierra el listado de valores entre llaves, las parejas de clave y valor se separan con comas, y la clave y el valor se separan con ':'.

```
dominios = {'ar': 'Argentina', 'es': 'España', 'tv': 'Tuvalu'}
```

En Python el tipo de dato asociado a los diccionarios se llama `dict`:

```
>>> type(dominios)
<class 'dict'>
```

Para declararlo vacío y luego ingresar los valores, se lo declara como un par de llaves sin nada en medio, y luego se asignan valores directamente a los índices.

```
materias = {}
materias["lunes"] = [6103, 7540]
materias["martes"] = [6201]
materias["miércoles"] = [6103, 7540]
materias["jueves"] = []
materias["viernes"] = [6201]
```

Para acceder al valor asociado a una determinada clave, se lo hace de la misma forma que con las listas, pero utilizando la clave elegida en lugar del índice.

```
>>> materias["lunes"]
[6103, 7540]
```

Atención

El acceso por clave falla si se provee una clave que no está en el diccionario:

```
>>> materias["domingo"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'domingo'
```

El operador `in` nos permite preguntar si una clave se encuentra o no en el diccionario:

```
>>> "lunes" in materias
True
>>> "domingo" in materias
False
```

Además podemos utilizar la función `get`, que recibe una clave k y un valor por omisión v, y devuelve el valor asociado a la clave k, en caso de existir, o el valor v en caso contrario.

```
>>> materias.get("lunes", [])
[6103, 7540]
>>> materias.get("domingo", [])
[]
```

Existen diversas formas de iterar los elementos de un diccionario. Por ejemplo, es posible iterar por sus claves y usar esas claves para acceder a los valores:

```
for dia in materias:
    print("El {} tengo que cursar {}".format(dia, materias[dia])
```

Es posible, también, obtener los valores como tuplas donde el primer elemento es la clave y el segundo el valor.

```
for dia, codigos in materias.items():
    print("El {} tengo que cursar {}".format(dia, codigos)
```

Recordar que el orden de los elementos no es relevante; por lo tanto no podemos asumir que el resultado de la iteración saldrá en ningún orden particular[1]. Además, no es posible obtener porciones de un diccionario usando `[:]`.

Hay muchas otras operaciones que se pueden realizar sobre los diccionarios, que permiten manipular la información según sean nuestras necesidades. Algunos de estos métodos pueden verse en la referencia al final de la unidad.

8.3 Algunos usos de diccionarios

Los diccionarios son una herramienta muy versátil. Se puede utilizar un diccionario, por ejemplo, para contar cuántas apariciones de cada palabra hay en un texto, o cuántas apariciones de cada letra.

Es posible utilizar un diccionario, también, para tener una agenda donde la clave es el nombre de la persona, y el valor es una lista con los datos correspondientes a esa persona.

[1] En versiones recientes de Python el orden de iteración corresponde al orden en que los elementos fueron añadidos al diccionario; en las versiones anteriores no se daba ninguna garantía acerca del orden de iteración.

Sabías que...

Los diccionarios en Python tienen una propiedad interesante: para cualquier diccionario D con N pares clave-valor, y para cualquier clave k, la operación D[k] es de *tiempo constante*, esto significa que, sin importar cuántos elementos tenga el diccionario, el tiempo de búsqueda de cualquier clave, es siempre aproximadamente igual.

Dado que las claves pueden ser de cualquier tipo inmutable (a diferencia de las listas, en las que los índices son números enteros entre 0 y $N-1$), para garantizar esta propiedad, el algoritmo utilizado para almacenar los datos en el diccionario debe ser más sofisticado que el utilizado para las listas.

Los diccionarios de Python están implementados usando una estructura de datos llamada *tabla de hash*. Para cada clave se calcula un valor numérico mediante un algoritmo llamado *código de hash*, que produce valores muy dispares dependiendo de la clave. Por ejemplo, el hash de la cadena `"Python"` es -539294296 mientras que el de `"python"`, una cadena que difiere en un caracter, es 1142331976. Los pares clave-valor del diccionario se guardan internamente en una lista, y el código de hash de la clave se utiliza para determinar el índice en la lista donde se ubicará cada par.

También podría utilizarse un diccionario para mantener los datos de los alumnos inscriptos en una materia, siendo la clave el número de padrón, y el valor una lista con todas las notas asociadas a ese alumno.

En general, los diccionarios sirven para crear bases de datos muy simples, en las que la clave es el identificador del elemento, y el valor son todos los datos del elemento a considerar.

Otro posible uso de un diccionario sería para realizar traducciones, donde la clave sería la palabra en el idioma original y el valor la palabra en el idioma al que se quiere traducir. Sin embargo esta aplicación es poco destacable, ya que esta forma de traducir suele dar resultados poco satisfactorios.

8.4 Resumen

- Los diccionarios son una estructura de datos muy poderosa, que permite almacenar un conjunto de pares *clave* → *valor*.
- Las claves deben ser inmutables y únicas.
- Los valores pueden ser de cualquier tipo, y pueden no ser únicos.
- El orden de los elementos no es relevante.

Referencia Python

`{clave1:valor1, clave2:valor2}`

Se crea un nuevo diccionario con los valores asociados a las claves. Si no se ingresa ninguna pareja de clave y valor, se crea un diccionario vacío.

`diccionario[clave]`

Accede al valor asociado con `clave` en el diccionario. Falla si la clave no está en el diccionario.

`clave in diccionario`

Indica si un diccionario tiene o no una determinada clave.

`diccionario.get(clave, valor_predeterminado)`

Devuelve el valor asociado a la clave. A diferencia del acceso directo utilizando `[clave]`, en el caso en que el valor no se encuentre devuelve el `valor_predeterminado`.

`for clave in diccionario:`

Permite recorrer una a una todas las claves almacenadas en el diccionario.

`diccionario.keys()`

Devuelve una secuencia con todas las claves que se hayan ingresado al diccionario.

`diccionario.values()`

Devuelve una secuencia con todos los valores que se hayan ingresado al diccionario.

`diccionario.items()`

Devuelve una secuencia con tuplas de dos elementos, en las que el primer elemento es la clave y el segundo el valor.

`diccionario.pop(clave)`

Quita del diccionario la clave y su valor asociado, y devuelve el valor.

8.5 Ejercicios

Realizar los ejercicios de la Unidad 8 de la Guía de "Enunciados de ejercicios de práctica obligatoria y opcional".

Licencia y Copyright

Copyright © Rosita Wachenchauzer <rositaw@gmail.com>
Copyright © Margarita Manterola <margamanterola@gmail.com>
Copyright © Maximiliano Curia <maxy@gnuservers.com.ar>
Copyright © Marcos Medrano <mmedrano@fi.uba.ar>
Copyright © Nicolás Paez <nicopaez@computer.org>
Copyright © Diego Essaya <dessaya@gmail.com>
Copyright © Dato Simó <dato@net.com.org.es>
Copyright © Sebastián Santisi <s@ntisi.com.ar>

Adaptación para UNDAV:
Copyright © Adriana S. Galli <agalli@undav.edu.ar>
Copyright © J. Daniel Sabella Rosa <jsabellarosa@undav.edu.ar>

Esta obra se distribuye bajo la Licencia Creative Commons Atribución-CompartirIgual 4.0 Internacional.

Los íconos utilizados fueron diseñados por Freepik.

El logo de Python es una marca registrada de la Python Software Foundation.

La publicidad de Cacao Droste es de dominio público, y fue descargada de Wikipedia.