Introducción a la Programación

Aprendiendo a programar usando Python como herramienta

Universidad Nacional de Luján

Contenidos

Unidad 1

Algunos conceptos básicos

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

1.1 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 una secuencia de *instrucciones* paso a paso que le indican a una computadora cómo realizar una tarea dada. En la computadora uno puede modificar un programa de acuerdo a la tarea que quiere realizar.

Las instrucciones se deben escribir en un lenguaje que nuestra computadora entienda. Los lenguajes de programación son lenguajes diseñados 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. Por ejemplo, si alguien nos dice *"Comprá el collar sin monedas"*, no sabremos si nos pide que compremos el collar que no tiene monedas, o que compremos un collar y que no usemos monedas para 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. En la carrera aprenderemos a medir los recursos que nos demanda una solución, y empezaremos a buscar la solución menos demandante en cada caso particular.

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

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.

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**.

1.3.1 La terminal

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

Para abrir la terminal:

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

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

1.3.2 El intérprete interactivo de Python

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

La terminal debería mostrar algo como se ve en la Figura 1.2.

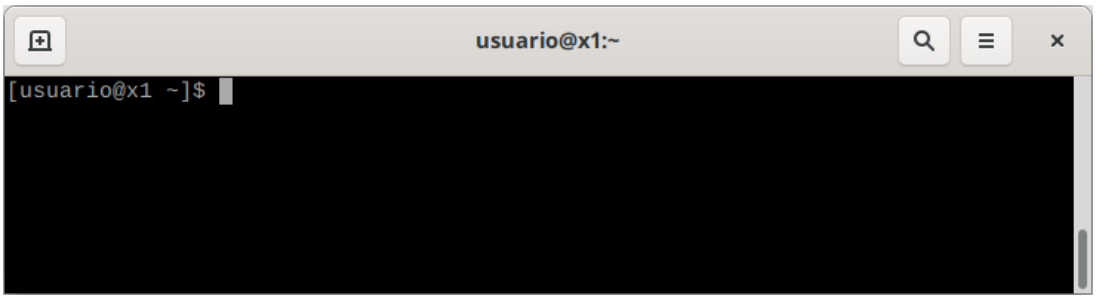


Figura 1.1: La terminal en un sistema operativo Linux.

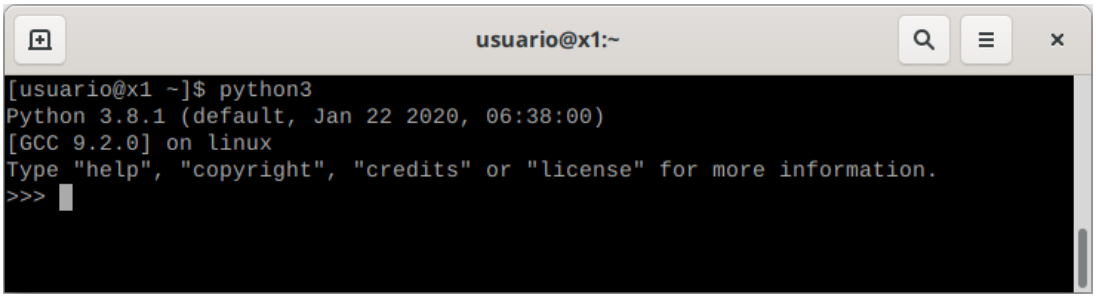


Figura 1.2: El intérprete de Python.

A partir de ahora vamos a mostrar el contenido de la terminal utilizando el siguiente formato:

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

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

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

Algunas sentencias sencillas, por ejemplo, permiten utilizar el intérprete como una calculadora simple con números enteros. Para esto escribimos la *expresión* que queremos resolver luego del *prompt* y presionamos la tecla Enter. El intérprete de Python evalúa la expresión y muestra el resultado en la línea siguiente. Luego nos presenta nuevamente el *prompt*.

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

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

```
>>> 5*7
```

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

1.4 Valores y tipos

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

Hay dos tipos de datos numéricos: los **números enteros** y los **números de punto flotante**. Los números enteros (`42`, `0`, `-5`, `10000`) representan el valor entero exacto que ingresemos. Los números de punto flotante (`5.3`, `-98.28109`, `0.0`)[1] son parecidos a la notación científica, almacenan una cantidad limitada de dígitos significativos y un exponente, por lo que sirven para representar magnitudes en forma aproximada. Según los operandos y las operaciones que hagamos usaremos la aritmética de los enteros o de los de punto flotante.

Vamos a elegir enteros cada vez que necesitemos recordar un valor exacto: la cantidad de alumnos, cuántas veces repito una operación, un número de documento, el dinero en una cuenta bancaria[2].

Cuando operamos con números enteros, el resultado es exacto:

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

Vamos a elegir punto flotante cuando nos interese más la magnitud y no tanto la exactitud, lo cual suele ser típico en la física y la ingeniería: la temperatura, el seno de un ángulo, la distancia recorrida, el número de Avogadro, el factorial de un número[3].

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

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

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

[2] ¿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.

[3] ¿Pero el factorial no es entero?, ¡sí!, pero si lo necesitamos, por ejemplo, para calcular un polinomio de Taylor, el factorial figura como denominador y ahí nos importa más su magnitud que su valor exacto.

1.5 Variables

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

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

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

1.6 Funciones

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

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

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

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

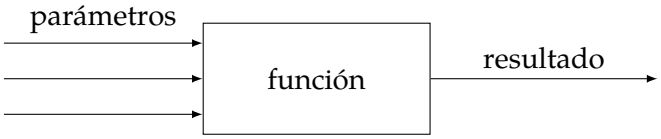


Figura 1.3: Una función recibe parámetros y devuelve un resultado.

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

1.7 Construir programas y módulos

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

Sabías que...

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

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

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

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

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

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

```
>>> 2 + 2
4
```

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

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

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

1.8 Interacción con el usuario

Ya vimos que la función `print` nos permite mostrar información al usuario del programa. En algunos casos también necesitaremos que el usuario ingrese datos al programa. Por ejemplo:

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

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

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

Solución.

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

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

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

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

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

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

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

```
1  print("Se calculará la suma de los cuadrados")
2
3  n1 = int(input("Ingrese un número entero: "))
4  n2 = int(input("Ingrese otro número entero: "))
5  suma = 0
6  suma = suma + n1*n1
7  suma = suma + n2*n2
8
9  print(suma)
10 print("Es todo por ahora")
```

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

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

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

En un programa Python podemos operar con cadenas de texto o con números. Las representaciones dentro de la computadora de un número y una cadena son 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
```

1.9 Estado y computación

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

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

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

Instrucción	Qué sucede	Estado
`print("Se calculará la suma de los cuadrados")`	Se despliega el texto "Se calculará la suma de los cuadrados" en la pantalla.	
`n1 = int(input("Ingrese un número entero: "))`	Se despliega el texto "Ingrese un número entero: " en la pantalla y el programa se queda esperando que el usuario ingrese un número.	

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

1.9.1 Depuración de programas

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

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

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

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

```
print("el valor de n2 es:", n2)

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

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

En este caso, la salida del programa será:

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

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

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.

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

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

   ⚠ Atención

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

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

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

- 1 pulgada = 0.0254 m

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

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

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

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

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

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

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

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

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

Mostrar por pantalla la variable metros
```

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

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

```
1 print("Convierte medidas inglesas a sistema metrico")
2
3 millas = int(input("Cuántas millas?: "))
4 pies = int(input("Y cuántos pies?: "))
5 pulgadas = int(input("Y cuántas pulgadas?: "))
6
7 metros = 1609.344 * millas + 0.3048 * pies + 0.0254 * pulgadas
8 print("La longitud es de ", metros, " metros")
```

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

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

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

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

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

```
False      class      finally    is         return
None       continue   for        lambda     try
True       def        from       nonlocal   while
and        del        global     not        with
as         elif       if         or         yield
```

```
assert     else       import     pass
break      except     in         raise
```

2.3.2 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 1.4, los números pueden ser tanto enteros (`0`, `111`, `-24`, almacenados internamente en forma exacta), como reales (`0.0`, `12.5`, `-12.5`, representados internamente en forma aproximada como números *de punto flotante*). Dado que los números enteros y reales se representan de manera diferente, se comportan de manera diferente frente a las operaciones. En Python, los números enteros se denominan `int` (de *integer*), y los números reales `float` (de *floating point*).
- Una expresión puede ser una *llamada a una función*: si `f` es una función que recibe un parámetro, y `x` es una variable, la expresión `f(x)` produce el valor que devuelve la función `f` al invocarla pasándole el valor de `x` por parámetro.

  Algunos ejemplos:
  - `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. También puede haber expresiones que sean una *cadena de caracteres* (letras, dígitos, símbolos, etc.), por ejemplo `"Ana"`.

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

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

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

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

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

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

> **Atención**
>
> No se pueden multiplicar cadenas entre sí
>
> ```
> >>> amiga * 3
> 'AnaAnaAna'
> >>> amiga * amiga
> Traceback (most recent call last):
>   File "<stdin>", line 1, in <module>
> TypeError: can't multiply sequence by non-int of type 'str'
> ```

2.3.4 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 :("
```

2.4 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).

Unidad 3

Funciones

3.1 Creación de funciones

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

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

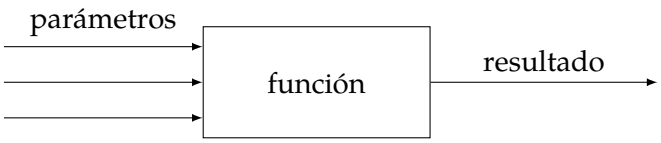


Figura 3.1: Una función recibe parámetros y devuelve un resultado.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Solución.

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

Para invocarla, deberemos hacer:

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

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

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

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

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

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

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

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

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

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

3.2 Documentación de funciones

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

Por convención, la documentación de una función se coloca en la primera línea del cuerpo de la misma, como una cadena de caracteres (que, como vimos en la sección 2.3.4, 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`:

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

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

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

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

3.3 Imprimir versus devolver

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

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

A continuación mostramos ambas implementaciones:

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

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

Veamos si funcionan:

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

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

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

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

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

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

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

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

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

3.4 Cómo usar una función en un programa

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

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

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

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

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

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

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

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

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

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

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

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

- El pseudocódigo final queda:

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

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

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

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

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

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

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

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

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

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

3.5 Alcance de las variables

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

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

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

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

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

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

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

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

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

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

3.6 Devolver múltiples resultados

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

Solución. La especificación es sencilla:

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

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

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

Esto es lo que sucede al invocar esta función:

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

Sabías que...

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

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

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

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

Atención

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

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

3.7 Módulos

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

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

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

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

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

```
import saludos

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

main()
```

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

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

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

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

3.7.1 Módulos estándar

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

Sabías que...

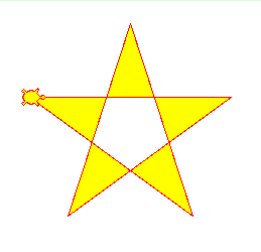
El lenguaje de programación *Logo*, creado en 1967 y utilizado principalmente con fines educativos, introdujo la idea de crear dibujos utilizando la metáfora de una *tortuga* que se mueve por la pantalla obedeciendo a comandos simples.

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

```
import turtle

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

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



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

3.8 Resumen

- Una función puede recibir ninguno, uno o más parámetros. Adicionalmente puede leer datos de la entrada del teclado.
- Una función puede no devolver nada, o devolver uno o más valores. Adicionalmente puede imprimir mensajes para comunicarlos al usuario.

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

Referencia Python

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

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

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

Documentación de funciones

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

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

`return valor`

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

`return valor1, valor2, valor3`

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

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

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

`import modulo`

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

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

`import modulo as variable`

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

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

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

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

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

Unidad 4

Decisiones

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

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

Diseñamos nuestra solución:

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

Es claro que la primera línea se puede traducir como

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

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

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

```
if <expresión>:
    <cuerpo>
```

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

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

4.1 Expresiones booleanas

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

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

4.1.1 Expresiones de comparación

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

De esta forma, `5 > 3` es una expresión booleana 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>`.

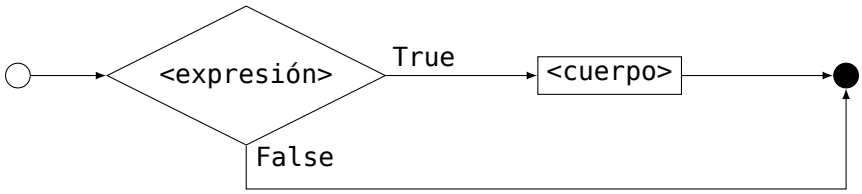


Figura 4.1: Diagrama de flujo para la instrucción `if`.

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`.

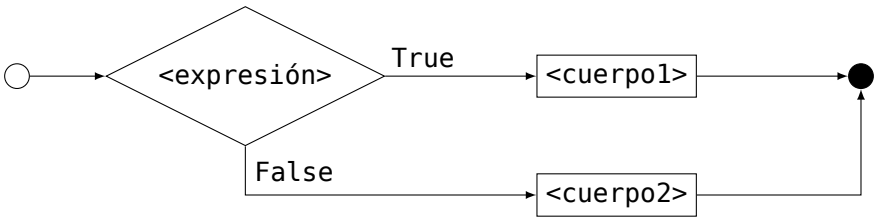


Figura 4.2: Diagrama de flujo para la estructura `if-else`.

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.

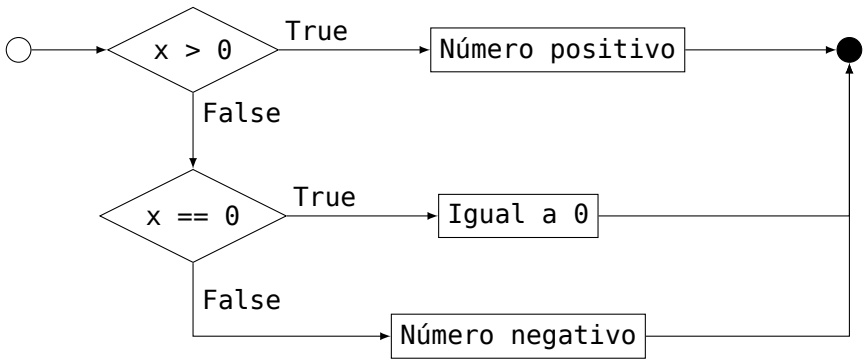


Figura 4.3: Diagrama de flujo para una estructura `if-elif-else`.

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`

Unidad 5

Ciclos

5.1 El ciclo definido

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

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

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

>>> suma_5_cuadrados()
55
```

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

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

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

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

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

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

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

Esta instrucción se lee como:

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

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

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

La secuencia de valores se puede indicar como:

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

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

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

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

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

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

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

Solución.

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

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

Supongamos ahora el siguiente problema:

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

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

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

Su ejecución es exitosa:

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

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

5.2 Ciclos indefinidos

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

Un ciclo indefinido es de la forma

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

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

El funcionamiento de esta instrucción es el siguiente:

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

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

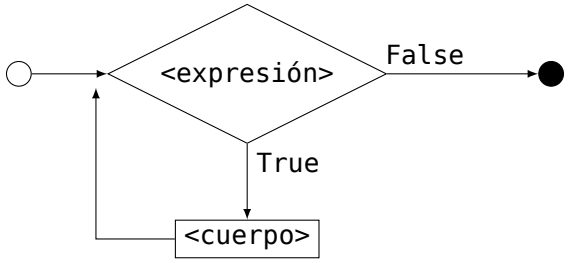


Figura 5.1: Diagrama de flujo para el ciclo indefinido `while`.

5.3 Ciclo interactivo

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

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

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

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

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

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

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

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

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

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

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

Encararemos la segunda opción:

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

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

El esquema del ciclo interactivo es el siguiente:

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

Ésta es una ejecución:

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

5.4 Ciclo con centinela

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

El esquema del ciclo con centinela es el siguiente:

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

```
    Realizar cálculos
    Pedir datos
```

El programa resultante es el siguiente:

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

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

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

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

Y ahora lo ejecutamos:

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

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

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

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

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

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

def muchos_pcn():
```

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

    centinela = leer_centinela()
```

Sabías que...

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

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

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

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

5.5 Resumen

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

Referencia Python

`for <nombre> in <expresión>:`

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

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

`while <condicion>:`

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

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

Unidad 6

Validación

6.1 Errores

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

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

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

6.2 Validaciones

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

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

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

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

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

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

6.2.1 Entrada del usuario

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

Por ejemplo, si se desea que un usuario ingrese un número entero, debemos comprobar el tipo de dato que ingresó. Python nos indica el tipo de una variable usando la función `type`.

```
def pedir_entero():
    """Solicita un valor entero y lo devuelve.
    Si el valor ingresado no es entero, imprime un mensaje de error y retorna el
    ↪ string "Error".
    """
    i = input("Ingrese un número entero: ")
    if type(i) is not int:
        print("El número no es de tipo entero")
        return("Error")
    else
        return int(i)
```

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

Sin embargo, esto no es satisfactorio: si el usuario no ingresa la información correctamente, el programa podría no continuar, si dicha información fuese necesaria para la resolución de la tarea del programa. Podemos hacerlo más *amigable* haciendo que se vuelva a pedir al usuario que ingrese la información:

```
def pedir_entero():
    """Solicita un valor entero y lo devuelve.
    Mientras el valor ingresado no sea entero, vuelve a solicitarlo."""
    while True:
        valor = input("Ingrese un número entero: ")
        if type(valor) is not int:
```

```
            print("'{}' no es un número entero.".format(valor))
        else
            return int(valor)
```

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

```
def pedir_entero():
    """Solicita un valor entero y lo devuelve.
    Si el valor ingresado no es entero, da 5 intentos para ingresarlo
    correctamente, y de no ser así, imprime mensaje
    de error de superación de intentos."""
    intentos = 0
    while intentos < 5:
        valor = input("Ingrese un número entero: ")
        if type(valor) is not int:
            print("'{}' no es un número entero.".format(valor))
        else
            return int(valor)
            intentos += 1
    print("Valor incorrecto ingresado en 5 intentos")
```

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

```
def lee_opcion():
    """Solicita una opción de menú y la devuelve."""
    while True:
        opcion = input("Ingrese A (Altas) - B (Bajas) - M (Modificaciones): ")
        if opcion.upper() in ("A", "B", "M"):
            return opcion
```

6.2.2 Comprobaciones por aserciones

Cuando queremos validar que los datos provistos a una porción de código contengan la información apropiada, ya sea porque esa información la ingresó un usuario, fue leída de un archivo, o porque por cualquier motivo es posible que sea incorrecta, es deseable comprobar que el contenido de las variables a utilizar estén dentro de los valores con los que se puede operar.

Estas comprobaciones no siempre son posibles, ya que en ciertas situaciones puede ser muy costoso corroborar las precondiciones de una función. Es por ello que este tipo de comprobaciones se realizan sólo cuando sea posible.

Por ejemplo, la función $\frac{1}{divisor}$ no está definida cuando el divisor es igual a 0. Es posible utilizar `assert` para comprobar las precondiciones de la función. La instucción `assert` de Python es una ayuda para la depuración del código ya que prueba una condición. Si la condición es verdadera, no hace nada y el programa simplemente continúa ejecutándose. Pero si la condición de aserción se evalúa como falsa, genera un error de ejecución con un mensaje de error opcional.

```
1 def division_de_1_por_divisor(divisor):
2     """Calcula el cociente entre 1 y el divisor.
3     Pre: el divisor debe ser un número distinto de 0
4     Post: se devuelve el valor del cociente entre 1 y el divisor
5     """
```

```
    assert isinstance(divisor, (int, float, complex)) and not isinstance(
    ↪ divisor, bool), "el divisor debe ser un número"
    assert divisor != 0, "el divisor no puede ser igual a 0"
    cociente = 1 / divisor
    return cociente
```

6.3 Resumen

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

Licencia y Copyright