# Nivel 1: Descubriendo el mundo de la programación



En este nivel se presentan en detalle los conceptos más básicos de cualquier lenguaje de programación. Aunque los conceptos se explican e ilustran usando el lenguaje de programación Python, la mayoría del contenido se aplica a muchos otros lenguajes de programación.

Los conceptos principales que se estudian en este nivel son los siguientes:

* Tipos de datos
* Variables
* Expresiones
* Operadores
* Funciones (definición e invocación)
* Parámetros

# Un programa para leer

En esta sección vamos a presentar un primer programa escrito usando el lenguaje de programación Python. Si nunca ha programado o si nunca ha programado en Python, probablemente habrá cosas que no va a entender completamente. Sin embargo, el simple hecho de revisar de forma detenida este programa le dará un poco de contexto para todos los elementos que se van a ir introduciendo a lo largo de este nivel.

## Por qué leer código de alguien más

Una habilidad muy importante para cualquier programador es la habilidad de leer código escrito por alguien más. Si dependemos de las explicaciones de alguien más para aprender un nuevo concepto o entender cómo utilizar una nueva herramienta o librería, será muy difícil que nos convirtamos en buenos programadores o el aprendizaje será muy lento. Por el contrario, la habilidad de leer el código de otros nos da una ventaja enorme y multiplica las fuentes de las que podemos aprender. En lugar de tener disponibles miles de libros, vamos a poder utilizar miles de millones de programas y de ejemplos que se encuentran disponibles en línea.

En general, cuando se lee código escrito por alguien más se deben tener en cuenta las siguientes recomendaciones:

* Hacer una lectura reflexiva, en varias pasadas. Es muy difícil asimilar con una sola lectura el objetivo general o los detalles de un programa que nunca habíamos visto. Además, los programas no se ejecutan en orden desde la primera línea hasta la última: como veremos más adelante, usualmente se definen bloques que se invocan desde otros lugares y que cambian completamente el orden de cualquier ejecución. La lectura se tiene que hacer entonces en varias pasadas, buscando diferentes cosas cada vez. Por otro lado, la lectura de código no es igual a la lectura de un texto cualquiera: se tiene que hacer de forma *lenta y cuidadosa*, evaluando constantemente qué es lo que se tiene claro y qué es lo que queda por descubrir.
* Buscar una comprensión global vs una comprensión de los detalles. No es lo mismo leer un programa buscando tener una idea global de cuál es su objetivo o cómo está estructurado, que leer un programa para entender en profundidad un detalle particular. Antes de empezar cada lectura se tiene que definir el objetivo que se va a buscar.
* Hacer explícitas las cosas que se desconocen o son poco claras. A medida que se vaya haciendo la lectura aparecerán cosas desconocidas, muchas de las cuales se resolverán más adelante en el mismo programa. Es una buena idea entonces ir marcando cuáles son esas cosas desconocidas y luego hacer explícitos los puntos en los que se resuelvan esas preguntas.
* Identificar y apropiar estrategias para resolver problemas. A medida que aumenta su experiencia, cada programador va construyendo en su cabeza su propio conjunto de estrategias que aplicará para resolver problemas aparezcan frecuentemente. Debido a esto, dos programadores pueden llegar a soluciones *aparentemente* muy diferentes incluso aunque estén usando el mismo lenguaje y los mismos algoritmos. Leer con cuidado el código escrito por alguien más nos abre las puertas al conjunto de estrategias de otros programadores y deberíamos aprovecharlo para enriquecer nuestra propia caja de herramientas.
* Diferenciar estilos. Cada programador tiene también un estilo propio que utiliza en la construcción de sus programas y que afecta la forma en la que se ven. Al leer programas escritos por alguien más se debería también observar con cuidado esas diferencias de estilo y evitar confundir diferencias de estilo (forma) con diferencias en las estrategias (fondo) o incluso diferencias algorítmicas.

## El primer programa en Python

El siguiente es un programa **completo** en Python que utiliza una buena parte de los conceptos que se presentarán dentro de este nivel. Cuando haya terminado el nivel podrá regresar a este programa para volver a estudiarlo: ¡Todo debería ser claro en esta nueva oportunidad!

**Instrucciones:** Lea el siguiente programa con cuidado, intentando entender qué es lo que está haciendo.

* Observe los bloques en los que está dividido y tenga en cuenta que en el lenguaje Python la *indentación* (la cantidad de espacios en blanco al inicio de cada línea, la sangría) es importante y sirve para definir bloques.
* Note que hay algunas palabras que se repiten.
* Nota también que hay algunas cosas escritas en español y otras escritas en inglés.
* Intente descubrir en qué orden se van a ejecutar cada una de las instrucciones del programa. *Ayuda:* Cada instrucción en este programa se va a ejecutar exactamente una vez.
* Anote qué cosas del programa no sabe qué significan.

def perimetro\_triangulo(cateto1: float, cateto2: float)->float:  
 hipotenusa = calcular\_hip(cateto1, cateto2)   
 return cateto1 + cateto2 + hipotenusa  
  
  
def calcular\_hip(cateto1: float, cateto2: float)->float:  
 suma\_cuadrados = (cateto1 \*\* 2) + (cateto2 \*\* 2)  
 hipotenusa = pow(suma\_cuadrados, 0.5)  
 return hipotenusa  
  
  
cadena\_cat\_1 = input("Indique la longitud del primer cateto: ")  
cadena\_cat\_2 = input("Indique la longitud del segundo cateto: ")  
cat\_1 = float(cadena\_cat\_1)  
cat\_2 = float(cadena\_cat\_2)  
  
perimetro = perimetro\_triangulo(3,4)  
  
print("El perímetro de un triángulo rectángulo que tenga catetos de longitud",  
 cat\_1, "y", cat\_2, "es", perimetro)

A continuación, se encuentra el mismo programa con una importante diferencia: esta vez se han incluido **comentarios**, es decir anotaciones que el programador dejó para que otras personas que quieran estudiar o modificar el programa puedan hacerlo con más facilidad.

**Escribir** buenos comentarios en el código (cortos, útiles y precisos) y **leer** con cuidado los comentarios de otros son muy buenas prácticas de programación.

# Este programa está escrito en el archivo perimetro.py  
  
def perimetro\_triangulo(cateto1: float, cateto2: float)->float:  
 """  
 Esta función calcula el perímetro de un triángulo rectángulo  
 dada la longitud de sus dos catetos  
 Parámetros:  
 cateto1 (float): la longitud del primer cateto del triángulo  
 cateto2 (float): la longitud del segundo cateto del triángulo  
 Retorno  
 (float): la longitud del perímetro del triángulo  
 """  
 # Usar la función calcular\_hip para calcular la longitud del lado faltante  
 hipotenusa = calcular\_hip(cateto1, cateto2)  
   
 # Sumar los tres lados y convertirlos en la respuesta de la función  
 return cateto1 + cateto2 + hipotenusa  
  
  
def calcular\_hip(cateto1: float, cateto2: float)->float:  
 """  
 Esta función calcula la longitud de la hipotenusa en un triángulo rectángulo  
 dada la longitud de sus dos catetos  
 Parámetros:  
 cateto1 (float): la longitud del primer cateto del triángulo  
 cateto2 (float): la longitud del segundo cateto del triángulo  
 Retorno  
 (float): la longitud de la hipotenusa  
 """  
 # Sumar la longitud de los catetos elevados al cuadrado  
 suma\_cuadrados = (cateto1 \*\* 2) + (cateto2 \*\* 2)  
   
 # Calcular la raiz cuadrada de la suma usando la función pow y el exponente 0.5  
 hipotenusa = pow(suma\_cuadrados, 0.5)  
 return hipotenusa  
  
  
# Solicitarle al usuario la longitud de los dos catetos  
cadena\_cat\_1 = input("Indique la longitud del primer cateto: ")  
cadena\_cat\_2 = input("Indique la longitud del segundo cateto: ")  
  
# Convertir los caracteres dados por el usuario en un número decimal  
cat\_1 = float(cadena\_cat\_1)  
cat\_2 = float(cadena\_cat\_2)  
  
# Llamar a la función con los valores recibidos  
perimetro = perimetro\_triangulo(3,4)  
  
# Mostrarle al usuario el resultado de la operación  
print("El perímetro de un triángulo rectángulo que tenga catetos de longitud", cat\_1, "y", cat\_2, "es", perimetro)

**Preguntas:** A partir de su lectura del programa, intente responder las siguientes preguntas. No se preocupe si no está seguro de algo, al final del nivel todas sus dudas deberían haber quedado aclaradas.

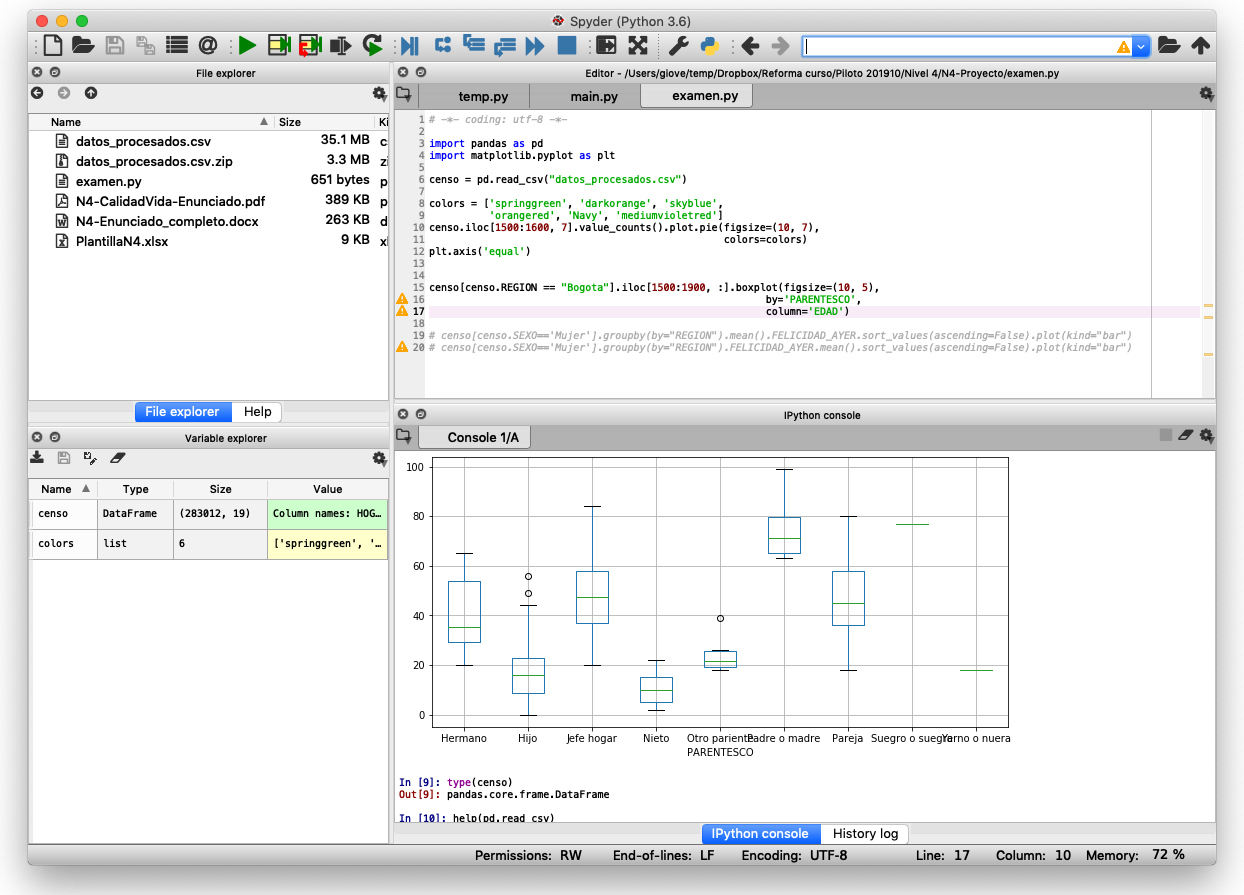
* ¿Cuál es el objetivo del programa?
* ¿Qué información tendrá que suministrar el usuario que ejecute el programa?
* ¿Cuál es el objetivo de cada bloque?
* ¿Qué es lo que primero se ejecuta?
* ¿Cual es la diferencia entre las cosas que están escritas en español y las que están escritas en inglés?

# Ambiente básico de trabajo

Para escribir programas es necesario utilizar unas herramientas que pueden variar dependiendo del lenguaje de programación. Incluso, para el mismo lenguaje es normal que existan muchas alternativas: no es necesario conocerlas todas, pero sí es importante poder utilizar al menos una con destreza.

En el caso de Python, lo usual es que se utilicen los siguientes elementos.

## IDE - Integrated Development Environment



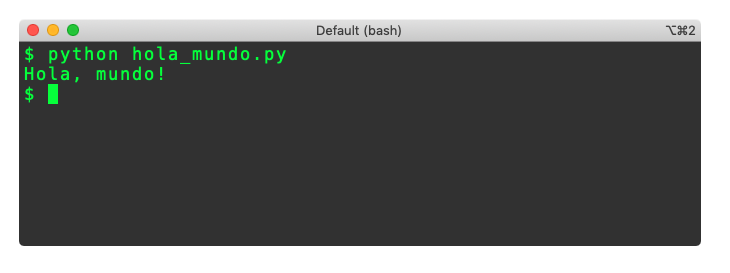
Captura de pantalla de Spyder

Un ambiente integrado de desarrollo es un programa que integra muchas herramientas que se requieren para programar con facilidad. El elemento más importante de un IDE es un editor enriquecido para el lenguaje, que es capaz de marcar errores de sintaxis y que utilice colores, entre otras ayudas, para facilitar la comprensión del código. Otras herramientas que se encuentran usualmente en un IDE son un depurador (para seguir la traza de una ejecución), un explorador de archivos y un mecanismo para ejecutar los programas con facilidad.

En este libro utilizaremos un IDE para Python llamado Spyder que, aunque no es el IDE más poderoso disponible, tiene muchas características que lo hacen propicio para aprender a programar:

* Es sencillo. Comparado con otros IDEs, ofrece menos opciones, pero esto hace que un desarrollador sin experiencia no se pierda en medio de muchas opciones que no sabría utilizar.
* Ayuda al desarrollo, pero no demasiado. Otros IDEs tienen muchos más mecanismos automatizados que sugieren o incluso cambian cosas a medida que el desarrollador va trabajando. Aunque esto es muy útil para desarrolladores experimentados, hemos visto que a los estudiantes les da una falsa sensación de que saben lo que están haciendo, cuando en realidad es el IDE el que los está guiando. El resultado de esto es que después no logran explicar lo que hicieron o no logran utilizar otra herramienta que los ayude de forma diferente.
* Multiplataforma. Está disponible para todas las plataformas principales (Windows, Linux, Mac)
* No usa formatos propietarios. Lo que se desarrolle en Spyder se puede llevar a otra herramienta sin ningún problema.
* Es gratuito y fácil de instalar.

## Intérprete

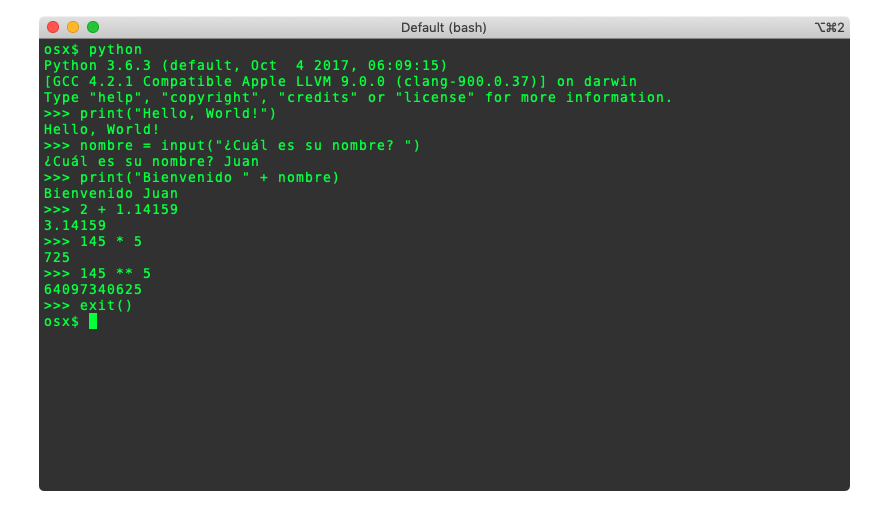


Interprete

Python es un lenguaje interpretado [[1]](#footnote-1): esto significa que para correr los programas escritos en Python es necesario que otro programa llamado intérprete los ejecute. En la imagen anterior se puede ver que se corrió un programa que se escribió en el archivo hola\_mundo.py usando el intérprete de Python.

Cada lenguaje de programación interpretado tiene su propio intérprete, e incluso puede haber varios intérpretes diferentes para el mismo lenguaje. En este libro vamos a suponer que usted está usando el intérprete básico, pero para Python hay varios adicionales que tienen características especiales como ser más rápidos, o requerir menos memoria, o incluso correr en plataformas especiales como plataformas de prototipado electrónico (ej. ESP8266 y ESP32) [[2]](#footnote-2).

## REPL para Python



REPL

Como en el caso de otros lenguajes interpretados, Python tiene ofrece una herramienta de tipo **REPL**, la cual permite que un usuario interactúe con el lenguaje y vaya ejecutando instrucciones una por una. La sigla REPL hace referencia al orden en el que se van realizando las operaciones:

* **Read**. En primer lugar, la herramienta lee lo que escribió el usuario y le informa si hay algún error.
* **Evaluate**. Luego, la herramienta evalúa lo que escribió el usuario usando el intérprete del lenguaje. Esto quiere decir que en este punto se ejecuta lo que el usuario haya escrito.
* **Print**. Se imprime en la herramienta el resultado de la ejecución para que el usuario lo pueda leer.
* **Loop**. Se repite el proceso completo.

En la imagen anterior se demuestra el uso del REPL de Python con varios ciclos de ejecución. Cada vez que aparecen los caracteres >>> se le pidió al usuario que ingresara un comando. Lo que aparece en la siguiente línea es el resultado de cada una de las ejecuciones.

Para acceder al REPL de Python hay dos opciones básicas:

1. Ejecutar el comando python desde la línea de comandos o el terminal (ver siguiente sección).
2. Usar el IDE. En el caso de Spyder, hay una ventana con el título 'IPython Console' que nos permite interactuar directamente con el REPL.

**Actividades:**

1. Abra el REPL en su computador, copie las instrucciones del ejemplo y revise que el resultado sea similar.
2. Evalúe en el REPL la instrucción 10/3. ¿Qué piensa del resultado? ¿Es el que usted esperaba?
3. Escriba la instrucción que convierta 15 grados Celsius al equivalente en grados Fahrenheit. Recuerde que cada grado Fahrenheit equivale a 5 novenos de un grado Celsius y que la escala está desplazada 32 grados. Ayuda: 0 grados Celsius equivalen a -32 grados Fahrenheit, 37.5 (la temperatura aproximada de un cuerpo humano) equivalen a 99.5, y 15 grados Celsius equivalen a 59 grados Fahrenheit.

## Línea de comandos / terminal / consola

La última herramienta que probablemente tenga que usar cuando esté programando es la línea de comandos del sistema operativo. Esto también se conoce como la terminal o la consola y es un ambiente *no gráfico* interactivo que le permite ejecutar comandos directamente sobre el sistema operativo. Entre otras muchas opciones, desde la línea de comandos usted puede ejecutar programas, trabajar con el sistema de archivos (crear, renombrar, mover, eliminar y hasta editar archivos) y utilizar una gran cantidad de utilidades.

Para los que nunca la han utilizado, usar la línea de comandos suele parecer incómodo y difícil. La realidad es que su uso eficiente requiere un tiempo de práctica, pero después termina siendo mucho más rápido para hacer ciertas tareas que utilizar un ambiente gráfico y el mouse. Por ejemplo, imagine que en una carpeta suya hubiera una colección de 500 fotos y que usted quisiera tener versiones reducidas de esas imágenes (*previews*, *thumbnails*). Normalmente a usted le tomaría una buena cantidad de clics abrir cada una de esas fotos y cambiarle el tamaño. Desde la línea de comandos de OS X usted simplemente puede usar el comando sips -Z 120 \*.png y realizar la operación sobre todas las imágenes.

El objetivo de esta sección no es explicar en detalle el funcionamiento de la línea de comandos de cada sistema operativo, sino mostrarle que existen y exhortarlo para que estudie por su cuenta su funcionamiento.

En Windows, la línea de comandos se invoca corriendo el programa cmd. 

En OS X, la línea de comandos se invoca corriendo el programa Terminal. 

En Linux, la línea de comandos suele estar corriendo todo el tiempo, pero dependiendo de la distribución que esté usando se llega a ella de formas diferentes.

# Valores y tipos de datos

El objetivo de esta sección es introducir y discutir los conceptos de *valor*, *literal*, y *tipo de datos*. Trabajaremos además con los tipos básicos *int*, *str* y *float* de Python.

La razón de ser de cualquier programa es poder manipular, mostrar, calcular o guardar *valores*. Estos valores podrían representar cosas que existan en una realidad, como por ejemplo la temperatura en una ciudad, la cantidad de dinero en una cuenta, el nombre de una persona, o la fecha actual. La naturaleza de los valores hace necesario separarlos en categorías porque en muchos casos no tiene sentido operarlos entre ellos. Por ejemplo, si sabemos que vamos a hacer una operación de multiplicación, no tiene sentido que mezclemos números con palabras. Estas categorías que se usan para separar los valores usualmente reciben el nombre de *tipos de datos*.

La pregunta natural que debería responderse ahora es: ¿Cómo se describe un valor dentro de un programa? En Python, la respuesta es que hay dos mecanismos básicos. El primero es a través de un *literal*, es decir que el valor se describe de forma directa siguiendo unas reglas dadas por el lenguaje. Por ejemplo, cuando en Python el literal 3 corresponde al valor entero *3*, el literal 3.14 corresponde al valor decimal *3.14* y el literal 'Saludos' corresponde a la palabra *Saludos*. El segundo mecanismo para expresar un valor es a través de una *expresión* que tiene que ser evaluada de alguna forma para averiguar su valor. Algunas expresiones válidas en Python son 1 + 2.22, round(3.14, 2) y 'Hola, ' + 'Mundo!'. No se preocupe por entender estos ejemplos: las siguientes secciones se dedicarán a explicarlos en detalle.

Python ofrece mecanismos para representar, interpretar y hacer operaciones sobre valores de varios tipos. Los más importantes son los que vamos a estudiar en esta sección: números enteros (int), números decimales (float) y cadenas de caracteres (str).

Adicionalmente, Python ofrece la función type que nos permite consultar de qué tipo es un determinado valor. Usaremos ahora esta función en tres ejemplos muy sencillos para introducir lo que se va a presentar en el resto de la sección y para observar el funcionamiento de la función misma.

>>> type(9)  
<class 'int'>  
>>> type(3.14)  
<class 'float'>  
>>> type('Hola')  
<class 'str'>

Usted puede probar estas mismas instrucciones escribiendo el ambiente de Python las instrucciones que se encuentran después de los caracteres >>>.

## Números enteros (int)

El primer tipo que vamos a estudiar es el que nos permite representar números enteros, positivos y negativos, y que en Python recibe el nombre de int.

En general, los números enteros se describen usando los literales a los que estamos acostumbrados: 0, 1, 2, 3, etc. Un número entero también puede ser negativo, así que los siguientes literales también son válidos: -1, -2, -3, etc. A diferencia de otros lenguajes, en Python no hay límites sobre los números enteros, así que cualquier número debería poder representarse sin problema [[3]](#footnote-3).

**Actividades:**

1. Abra el REPL en su computador y escriba varios números enteros para verificar que sean válidos en Python. Use la función type para comprobar que sean de tipo int.
2. Compruebe usando un número muy grande (varias decenas de dígitos) que Python usa el tipo int incluso para números muy grandes.

## Números decimales (float)

Como complemento al tipo int, Python también ofrece el tipo float que permite representar números con decimales. Es importante notar que, debido a restricciones técnicas, Python frecuentemente tiene que redondear números decimales. A manera de ejemplo, consideremos el resultado de hacer la división 10/3 que resulta en 3.3333333333333335 en lugar del resultado esperado.

En Python, los literales para representar números decimales utilizan un punto como separador. Es decir que los siguientes son todos números de tipo float: 0.0, -1.1, 2.33, -4.5555557.

Una característica interesante de Python que no está presente en muchos otros lenguajes es la conversión automática que hace de enteros a decimales, especialmente cuando se hacen operaciones de división.

**Actividades:**

1. Abra el REPL y pruebe ahora a escribir números con decimales. Utilice la función type para verificar el tipo obtenido.
2. Escriba expresiones aritméticas para evaluar y revise el tipo resultado.
3. Escriba números con una gran cantidad de decimales y revise qué pasa con el número cuando se hace la evaluación.

## Cadenas de caracteres (str)

Las cadenas de caracteres son un tipo de dato muy importante dentro de Python y se denotan como de tipo str. Una cadena de caracteres es mucho más que una palabra. Una cadena de caracteres es una secuencia de símbolos que puede incluir letras (minúsculas y mayúsculas), números, signos de puntuación, espacios y hasta *emojis*. Es decir que una cadena de caracteres puede servirnos para representar cosas como un nombre, un número serial o un texto completo.

En Python hay 3 formas de representar literales que sean de tipo str.

1. Rodeándolos con comillas *sencillas*. Es decir que los siguientes son 3 ejemplos de cadenas de caracteres válidas: 'abc', 'a1 b2 c3', '¡Hola, Mundo!'.
2. Rodeándolos con comillas *dobles*. Es decir que los siguientes son 3 ejemplos de cadenas de caracteres válidas: "abc", "a1 b2 c3", "¡Hola, Mundo!".
3. Rodeándolos con *tres comillas* sencillas o dobles. Es decir que los siguientes son 3 ejemplos de cadenas de caracteres válidas: """abc""", '''a1 b2 c3''', """¡Hola, Mundo!""".

Puede parecer una exageración tener 3 opciones diferentes, pero hay motivos claros para esto. Consideremos por ejemplo una cadena con un texto en inglés que tenga el siguiente valor: It's today!

Si quisiéramos representar la cadena usando comillas sencillas sí tendríamos un problema: ¿Si el literal fuera 'It's today!' cómo podría hacer Python para saber que la cadena termina en la tercera comilla y no en la segunda? La solución más fácil para este problema es representar la misma cadena usando comillas dobles: "It's today!". De esta forma se elimina la ambigüedad sin tener que recurrir a soluciones más complicadas.

De forma similar, si nuestra cadena tuviera comillas dobles dentro de ella, el literal se podría escribir con comillas sencillas y también se resolvería el problema.

Desafortunadamente este truco no funciona cuando la cadena incluye comillas dobles y comillas sencillas. Por ejemplo, una cadena con el siguiente valor tendría este problema: She said to me "That's mine!".

La solución en este caso es utilizar expresiones especiales para representar las comillas dobles o las comillas sencillas. Es decir que en lugar de representar una comilla sencilla dentro de la cadena usando el caracter ' se usaría la expresión \'. También existe la expresión equivalente \" para las comillas dobles. Esto quiere decir que el literal para la cadena del ejemplo podría ser 'She said to me "That\'s mine!"' o "She said to me \"That\'s mine!\"".

Veamos ahora la opción de las tres comillas sencillas o dobles, que resuelve una limitación importante que tienen las otras dos opciones: cuando se usan tres comillas, las cadenas pueden tener cambios de línea dentro de los literales. Considere el siguiente fragmento de código válido en Python que se visualiza tal como fue tecleado en el intérprete del lenguaje [[4]](#footnote-4).

>>> """one foolish heart  
... five wits unswayed  
... a thousand errors note"""

Por el contrario, si se usara una sola comilla sencilla o una sola comilla doble para iniciar el literal, se produciría un error como el siguiente al terminar la primera línea:

>>> 'one foolish heart  
 File "<stdin>", line 1  
 'one foolish heart  
 ^  
SyntaxError: EOL while scanning string literal

Esto no quiere decir que una cadena descrita usando los dos primeros métodos no pueda tener cambios de línea. Lo que pasa es que en este caso se debe utilizar una expresión especial para representar ese cambio de línea: \n. Esta expresión se conoce como un caracter de control y es utilizada en la mayoría de lenguajes de programación para hacer referencia a un cambio de línea al final de un párrafo (es lo que su teclado envía cuando usted presiona la tecla *enter*).

Veamos entonces cómo se usaría dentro de un literal:

>>> 'one foolish heart\nfive wits unswayed\na thousand errors note'

**Actividades:**

1. Practique en el REPL la definición de cadenas utilizando cada uno de los métodos descritos.

## Conversiones entre tipos de datos

En Python es posible hacer conversiones entre diferentes tipos de datos para convertir, por ejemplo, una cadena en un número decimal, o un entero en una cadena. Esto sólo puede hacerse cuando tenga sentido y es útil para poder utilizar operadores de otros tipos de datos. Por ejemplo, no podemos convertir la cadena 'abc' en un entero, pero sí podemos convertir la cadena '3.4' en un número decimal para después sumarlo al valor 4.55.

En una de las próximas secciones estudiaremos más en detalle las funciones de conversión, pero por ahora usted debe saber que existen y cuál es su objetivo principal:

* int(x): convierte el valor x a un entero. Por ejemplo, convierte el número 3.14 a 3 y la cadena '-4' a 4.
* float(x): convierte el valor x a un número decimal. Por ejemplo, convierte el número entero 3 a 3.0 y la cadena '-4.5' a 4.5.
* str(x): convierte el valor x a una cadena de caracteres. Por ejemplo, convierte el número entero 3 a la cadena '3' y al número decimal -4.5 a la cadena '-4.5'.

**Actividades:**

1. Use el REPL para convertir entre todos los tipos de datos:
   * Convierta un entero a un flotante
   * Convierta un flotante a un entero
   * Convierta una cadena a un entero
   * Convierta una cadena a un flotante
   * Convierta un entero en una cadena
   * Convierta un flotante a una cadena

# Variables, expresiones y operadores

El objetivo de esta sección es introducir y discutir los conceptos de *instrucción*, *valor*, *variable*, *asignación*, *expresión* y *operador*.

Para empezar, lea con cuidado el siguiente programa escrito con el lenguaje de programación Python. Aunque breve, ilustra los conceptos más básicos del lenguaje, empezando por el concepto de **instrucción**.

v1 = 5  
v2 = 1 + 2  
v3 = v1 + v2  
v4 = v3 / (4 + 1)  
print("v4:", v4)

Este programa está compuesto por 5 instrucciones, cada una de las cuales se escribió en una línea aparte. Como veremos, cada una de estas instrucciones le da una orden al computador y esa orden se tiene que terminar de ejecutar antes de que se pueda pasar a la siguiente instrucción. Esto quiere decir que podemos estar seguros de que la última instrucción (print(v4)) se va a ejecutar después de que se hayan ejecutado las 4 instrucciones anteriores. ¡Aunque puede parecer simple, este concepto es extremadamente importante!

## Analizando la primera instrucción

Analicemos ahora la primea instrucción del programa:

v1 = 5

Esta instrucción se puede dividir en dos partes dividas por el caracter =. En la parte derecha, encontramos sólo el caracter 5, el cual expresa el **valor** que corresponde al número natural 5.

En la parte izquierda encontramos sólo el texto v1. Como sabemos, este texto no puede expresar ninguno de los tipos de datos que ya conocemos (*int*, *float*, *str*) así que v1 no puede ser un valor. En este caso, v1representa entonces el nombre de una **variable** definida por el programador.

Una **variable** es un espacio en la memoria del computador en el cual se puede almacenar un valor o del cual se puede leer un valor, mientras se ejecute el programa. Cuando un programador define una variable, le asigna un nombre o identificador para que sea fácil de recordar y utilizar en el resto del programa. En nuestro ejemplo, la variable tiene el nombre v1.

**Nota:** Se debe procurar que los nombres de las variables sean lo más descriptivos posibles para que su objetivo sea claro. En el ejemplo la variable tiene un nombre que no es muy explícito, pero en las siguientes líneas quedará claro por qué se seleccionó el nombre v1. En el programa completo del inicio del capítulo, una de las variables se llamaba hipotenusa: sería difícil encontrar un nombre que expresara con más claridad el objetivo de esa variable.

Finalmente, volvamos al caracter = que separa la *variable* que se encuentra a la izquierda del *valor* que se encuentra a la derecha. En Python, el caracter = se utiliza para especificar que en un programa se debe hacer una **asignación**. Es decir, cuando se ejecute una instrucción con una asignación, el programa debe tomar el valor que se encuentra a la **derecha** del caracter y lo debe almacenar en la variable que se encuentra a la **izquierda**.

La instrucción que estamos analizando entonces se encarga de crear una variable con el nombre v1 y almacenar en ella el valor entero 5. En Python, el tipo de una variable depende del valor que esté almacenado en ella así que en este caso la variable será de tipo **int** [[5]](#footnote-5).

**Cuidado**: Cuando se hace una asignación, el valor de la derecha se almacenará en la variable de la izquierda. Es un error bastante común entre principiantes escribir instrucciones como '5 = v1': esta instrucción no sería válida en Python y el intérprete mostraría un error similar al siguiente:

SyntaxError: can't assign to literal

## Operadores y expresiones

Si analizamos ahora la segunda línea del programa, veremos que tiene varios de los elementos que acabamos de estudiar:

v2 = 1 + 2

A la izquierda tenemos una variable llamada v2 a la cual le vamos a asignar el valor que se encuentra a la derecha. La diferencia es que en este caso el valor no está definido usando un literal sino usando una **expresión**: 1 + 2.

En Python, una **expresión** es una combinación de valores, literales, variables, llamados y operadores, que al evaluarse produce un valor. Cuando escribimos una instrucción como la del ejemplo, le estamos indicando a Python que la expresión de la derecha debería evaluarse para producir un valor y que ese valor debería almacenarse en la variable. En nuestro ejemplo, el valor de la expresión de la derecha será 3, es decir el resultado de usar el **operador** + a los dos literales **1** y **2**.

## Instrucciones con variables

A continuación, analizaremos las siguientes dos instrucciones del programa, pero recordando un principio muy importante: para que estas instrucciones se ejecuten, se tienen que haber ejecutado antes las anteriores. Es decir, que cuando estas instrucciones se ejecuten ya se habrá creado la variable v1 y se le habrá asignado el valor 5 y ya se habrá creado la variable v2 y se le habrá asignado el valor 3 [[6]](#footnote-6).

v3 = v1 + v2

Esta instrucción es muy similar a la anterior: a la nueva variable v3 se le debe asignar el valor resultado de evaluar la expresión de la derecha. La diferencia principal es que en este caso la expresión no está construida a partir de literales como 1 y 2, sino a partir de nombres de variables. Esto quiere decir que cuando llegue el momento de ejecutar la instrucción, Python tiene que evaluar la expresión de la derecha *antes* de poder hacer la asignación.

Ahora bien, ¿cómo puede calcular Python la suma de dos cosas que no son literales? La respuesta es que también tiene que evaluar esas dos cosas para averiguar qué valor tienen. Como v1 y v2 son variables, la evaluación es muy sencilla porque sólo requiere consultar cuál es el valor almacenado en esas variables. Eso quiere decir que, después de evaluar el valor de esas variables, Python está listo para hacer la siguiente asignación:

v3 = 5 + 3

En este punto volvimos a una situación idéntica a la de la instrucción anterior y ya sabemos que se va a resolver dejando el valor entero 8 en la variable v3.

**Nota:** La evaluación de la instrucción anterior fue exitosa porque a las variables v1 y v2 ya se les había asignado un valor anteriormente. Si en lugar de v1 hubiéramos escrito un nombre de variable inexistente, como vv, habríamos encontrado un error como el siguiente:

NameError: name 'vv' is not defined

## Paréntesis y tipos de datos

Antes de pasar a la siguiente instrucción, observemos lo que responde Python cuando le preguntamos por los tipos de las variables v1, v2 y v3.

>>> type(v1)  
<class 'int'>  
>>> type(v2)  
<class 'int'>  
>>> type(v3)  
<class 'int'>

No es una sorpresa que v1 sea de tipo int dado que la asignación se hizo con un literal. En el caso de v2 y v3 Python decidió que el tipo debía ser int dado que el valor se calculó a partir de la suma de valores de tipo int.

Pasemos ahora a la siguiente instrucción, en la que nuevamente tenemos una asignación y una expresión con operadores a la derecha.

v4 = v3 / (4 + 1)

Lo primero que tenemos que notar es que en este caso se utilizaron paréntesis, los cuales tienen el mismo efecto que tendrían si estuviéramos resolviendo un ejercicio de matemáticas. En este caso los paréntesis son obligatorios porque en Python los operadores matemáticos siguen las reglas de precedencia tradicionales. Es decir que la expresión v3 / (4 + 1) tiene un valor diferente al de la expresión v3 / 4 + 1 porque el operador de división (/) tiene mayor precedencia que el operador de suma (+). Debido a esto, lo primero que va a hacer Python cuando empiece a evaluar la expresión de la derecha es encontrar el valor para lo que estaba dentro de los paréntesis y reescribirá la instrucción como:

v4 = v3 / 5

Ahora lo que Python tiene que hacer es evaluar la expresión de la derecha, pero para eso primero debe consultar el valor de la variable v3. Después de hacer esto, la instrucción se reescribirá como:

v4 = 8 / 5

Finalmente tenemos una instrucción que sólo tiene literales y el operador de división, así que la expresión de la derecha se puede obtener simplemente calculando la división entre 8 y 5, para luego asignárselo a la variable v4.

Ahora bien, el operador de división en Python tiene la propiedad interesante de que su resultado es de tipo **float**. Es decir que independientemente del valor de v3, el valor de la variable v4 va a ser un número decimal. Esto lo podemos comprobar si le preguntamos a Python por el tipo de la variable:

>>> type(v4)  
<class 'float'>

## Otro tipo de instrucciones

Si revisamos con cuidado las 4 instrucciones que hemos analizado, nos daremos cuenta que lo único la única orden que le hemos dado al programa es que almacene valores dentro de variables. Como vamos a ver a continuación, la última instrucción de nuestro programa es fundamentalmente diferente a las anteriores porque no hace ninguna asignación.

print("v4:", v4)

En primer lugar, revisemos qué pasa cuando se ejecuta esta línea. Suponiendo, como siempre, que ya se habían ejecutado las líneas anteriores del programa, al ejecutar esta línea lo que debería ver el usuario en la consola es lo siguiente.

v4: 0.6

Lo que hace nuestra instrucción es una invocación a una función básica de Python llamada print y le pasa dos parámetros ( "v4:" y v4). Más adelante estudiaremos más a fondo la función print pero lo que nos interesa saber en este momento es que la función sirve para mostrarle al usuario los valores que hayamos usado como parámetros. Más adelante estudiaremos también cómo definir nuestras propias funciones.

En este caso la invocación a la función se está haciendo con dos parámetros que podemos identificar porque aparecen dentro de paréntesis y están separados por una coma. El primer parámetro ("v4:") es un literal de tipo str (cadena de caracteres). El segundo parámetro (v4) es el nombre de la variable a la que se le asignó un valor en la instrucción anterior. Note que como estamos haciendo referencia a una variable, no se utilizan comillas sencillas ni dobles.

Al igual que una asignación necesita calcular el valor de la derecha para que se pueda almacenar el valor en la variable, para hacer una invocación a una función se necesitan los valores de los parámetros. En nuestro caso, el primer parámetro es un literal así que su valor ya es conocido; como el segundo parámetro es una variable, Python buscará el valor que se había almacenado en la variable y lo usará para hacer la invocación.

El último paso en la ejecución de nuestro programa no podemos verlo: son las instrucciones que se encuentran dentro de la función print, las cuales hacen que aparezcan los valores de los parámetros en la consola. Como esta es una función nativa del lenguaje, las instrucciones que la implementan son parte del código fuente del intérprete del lenguaje y no son fáciles de encontrar.

## Comentarios en Python

El siguiente programa es exactamente equivalente al que hemos estado estudiando, con la diferencia de que se han incluido *comentarios*.

# Definir 4 variables (v1 hasta v4)  
v1 = 5  
v2 = 1 + 2  
v3 = v1 + v2 # El valor de v3 es de tipo int porque es la suma de dos int  
v4 = v3 / (4 + 1) # El valor de v4 va a ser de tipo float  
print("v4:", v4) # Mostrarle al usuario el valor de v4

Un comentario en un programa es una anotación que dejó el programador para que otros programadores, o él mismo, puedan entender con más facilidad el objetivo de un programa o de un bloque de código. En el caso de Python, la forma más común de incluir comentarios es utilizando el caracter #: todos los caracteres que se encuentren a la derecha de este caracter serán ignorados por el intérprete.

En el caso de nuestro ejemplo, los comentarios se ha incluido un número relativamente grande de comentarios para un programa tan sencillo. Sin embargo, esto sirve para ilustrar un principio importante: ante la duda, es mejor tener más comentarios que menos comentarios en un programa.

## Más operadores en Python

En las secciones anteriores estudiamos sólo los operadores para sumar y dividir. A continuación, describimos otras de las operaciones que es posible hacer en Python.

### Operadores para números

Los siguientes son los operadores disponibles para hacer operaciones con números. Es decir que estos operadores pueden aplicarse sólo sobre datos que sean de tipo **int** o **float**.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Operación | Operador | Aridad (1) | Precedencia (2) | Ejemplo | Valor |
| Exponenciación | \*\* | Binario | 1 | 10 \*\* 3 | 1000 |
| Identidad | + | Unario | 2 | + 2 | +2 |
| Cambio de signo | - | Unario | 2 | -(-2) | +2 |
| Multiplicación | \* | Binario | 3 | 10 \* 3 | 30 |
| División (3) | / | Binario | 3 | 10 / 3 | 3.3333 |
| División entera (4) | // | Binario | 10 / 3 | 3 |  |
| Módulo (5) | % | Binario | 3 | 10 % 3 | 1 |
| Suma | + | Binario | 4 | 10 + 3 | 13 |
| Resta | - | Binario | 4 | 10 - 3 | 7 |

(1) El término *aridad* hace referencia a la cantidad de operandos sobre los que se aplica el operador. Los operadores unarios sólo requieren un operador mientras que los binarios necesitan 2.

(2) La precedencia de un operador indica en qué orden se evaluarán varios operadores en caso de que no haya paréntesis que permitan resolver el orden. En Python, el primer operador que se evaluará es el operador de exponenciación. En caso de que la precedencia de dos operadores sea la misma, los operadores se aplicarán de izquierda a derecha. Por ejemplo, en el caso de la expresión 70 // 2\*\*3 / 4 primero se evaluará la operación de exponenciación (la de mayor precedencia), luego se evaluará la división entera y finalmente la división.

(3) En Python, la división siempre produce un resultado de tipo float, el cual muchas veces se redondea automáticamente. En el caso del ejemplo, la división 10/3 tiene como resultado el número decimal 3.3333333333333335, que es la mejor aproximación que puede hacer Python del resultado real de la división. El valor de la expresión 9/3, aunque podría ser un número entero (int), es el número decimal (float) 9.0.

(4) Para valores positivos, la operación de *división entera* calcula la parte entera del resultado de la división y por ende siempre es un número entero. En el caso de la expresión 10//3 el valor es el número entero (int) 3.

Para números negativos, el resultado es el mayor número entero menor o igual al resultado de la división. Es decir que para el valor de la expresión -10//3 es el número entero -4

(5) La operación módulo calcula el **residuo** de la división entera entre dos números. Por ejemplo, el valor de la expresión 10%3 es igual a 1 porque la división entera entre 10 y 3 tiene como resultado 3 con un residuo de 1 [[7]](#footnote-7).

La operación módulo es muy utilizada para averiguar la paridad de un número: el resultado de la expresión x%2 será 0 sólo cuando x sea un número par y será 1 cuando x sea impar.

### Operadores para cadenas de caracteres

Python sólo ofrece dos operadores que se pueden aplicar sobre cadenas de caracteres.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Operación | Operador | Aridad (1) | Precedencia (2) | Ejemplo | Valor |
| Concatenación | + | Binario | 1 | 'abc' + 'def' | 'abcdef' |
| Repetición | \* | Binario | 1 | 'ab' \* 3 | 'ababab' |

El operador de concatenación se utiliza para unir dos cadenas de caracteres y convertirlas en una sola.

El operador de repetición se aplica sobre dos operandos de diferente tipo: una cadena de caracteres y un entero. El resultado será una cadena de caracteres que será el resultado de repetir la cadena tantas veces como indique el número.

### Expresiones con múltiples tipos de datos

En Python no es posible escribir expresiones que combinen valores de diferentes tipos a menos que se hagan conversiones explícitamente. Por ejemplo, para construir una cadena que combine caracteres y números se debe hacer una conversión como en el siguiente ejemplo:

'El salón asignado es el ' + str(614) + ' del edificio ML'

Por otro lado, para hacer operaciones numéricas con valores almacenados en una cadena se debe convertir esa cadena a un número como en el siguiente ejemplo:

1 + float('2.3') + 3

Finalmente, si se quieren hacer combinaciones de tipos se deben utilizar paréntesis que eliminen los posibles problemas.

'El resultado es ' + str(2\*\*3)

En este siguiente ejemplo lo primero que se hace es evaluar la operación de exponenciación, luego se convierte el resultado a una cadena y finalmente se concatenan las dos cadenas.

## Ejercicios

1. ¿Cuáles de las siguientes líneas no son instrucciones válidas en Python? (suponga que las instrucciones se van ejecutando una después de la otra)
   * variable = 5
   * 5 = variable
   * var1 = var2 + 5
   * var1 = var1 + 5
   * var = var + 5
2. ¿Qué resultados se obtendrán al evaluar las siguientes expresiones en Python? Verifique los resultados evaluando las expresiones en el intérprete de Python.
   * 2 + 3 + 1 + 2
   * 2 + 3 \* 1 + 2
   * (2 + 3) \* 1 + 2
   * (2 + 3) \* (1 + 2)
   * +---6
   * -+-+6
   * -3 / 2 - 1
   * -3 // 2 - 1
   * 3 % 2 - 1
3. ¿Qué valor se mostrará en la pantalla después de ejecutar el siguiente código?

* z = 1  
  z += 2  
  z \*= 2  
  z //= 2  
  z -= 2  
  z %= 2  
  z \*\* 2  
  z /= 2  
  print(z)

1. ¿Qué resultado se obtendrá al evaluar la siguiente expresión en Python?

* 'a' \* 3 + '/\*' \* 5 + 2 \* 'abc' + '+'

1. ¿Qué resultado se obtendrá al evaluar las siguientes expresiones en Python?

* 25 / 3 // 2  
  25 / (3 // 2)  
  (25 / 3) // 2  
  25 // 3 / 2  
  25 // (3 / 2)  
  (25 // 3) / 2

## Más allá de Python

Esta sección presentó en detalle los detalles más importantes de la sintaxis de Python. Sin embargo, los conceptos presentados son comunes a la mayoría de lenguajes de programación *imperativos* [[8]](#footnote-8). Aunque puede parecer que las diferencias entre Python y otros lenguajes son muy grandes, en realidad los conceptos que se manejan son básicamente los mismos: todos los lenguajes tienen literales, tienen variables para almacenar temporalmente valores, tienen expresiones que se tienen que evaluar, y tienen mecanismos para invocar fragmentos de código definidos en algún otro lugar.

Si tenemos perfectamente claro todo lo expuesto en este capítulo, aplicar las mismas ideas a otros lenguajes debería ser muy sencillo.

# Funciones

El objetivo de esta sección es introducir y discutir los conceptos de *función* y *parámetro*, e ilustrarlos con las funciones básicas más importantes del lenguaje.

En esta sección estudiaremos el concepto de **función**, que en Python es el principal mecanismo para estructurar un programa: en lugar de escribir todas las instrucciones de un programa una después de la otra en un mismo archivo, usando funciones es posible organizar las instrucciones en bloques de código que tengan un objetivo preciso y que puedan ser reutilizados.

Por ahora introduciremos los conceptos principales y los ilustraremos usando funciones que son parte de Python. La siguiente sección mostrará como definir nuevas funciones.

## El concepto de función

En términos matemáticos, una función es una relación entre valores de tal forma que para cada combinación de los valores *de entrada* haya sólo un valor *de salida*. Además, las funciones se suelen describir usando una fórmula, de tal forma que dados unos valores de entrada sea sencillo calcular el valor de salida.

A manera de ejemplo, consideremos la siguiente definición de una función:

f(x, y) = x y

Esta definición expresa que existe una función llamada *f* y cuyo valor depende de dos parámetros llamados *x* y *y*: cuando se evalúa la función, asignándole valores específicos a *x* y a *y*, el valor de *f* será igual al valor de calcular *x elevado a la y*. Por ejemplo, si evaluamos la función con los valores *x=2* y *y=3* el resultado será el número *8*. También sabemos que si el valor de *y* es 0, entonces el valor de evaluar la función siempre será *1* independientemente del valor de *x*. Finalmente es importante notar que el resultado de evaluar *f(x,y)* será siempre el mismo para cada combinación de valores de *x* y *y*: no es posible que *f(2,3)* algunas veces tenga el valor *8* y en otras ocasiones tenga un valor diferente.

Una función tiene entonces un nombre, que en el ejemplo anterior era *f*, y un conjunto de parámetros[[9]](#footnote-9) a los que se les deben dar valores cuando se quiera evaluar la función. En lo posible, los nombres de las funciones y sus parámetros deberían ayudar a aclarar el objetivo de una función. Por ejemplo, la función anterior podría reescribirse de la siguiente manera y sería mucho más legible:

potencia(base, exponente) = base exponente

## Funciones en Python

El concepto matemático de función presentado en la sección anterior fue adoptado por Python, pero con unas pequeñas modificaciones que explicaremos en la siguiente sección. Por ahora, lo importante es que en Python es posible definir funciones dándoles un nombre, especificando sus parámetros, y explicando cómo se debe calcular un valor específico a partir de los parámetros. Como veremos en esta sección, Python cuenta con un gran número de funciones pre-definidas que podemos utilizar en nuestros programas. En la próxima sección estudiaremos cómo podemos hacer para definir nuestras propias funciones.

La acción más interesante que podemos hacer sobre una función es *invocarla*. Esto es lo mismo que *evaluarla* para poder saber cuál es sería su valor dados valores específicos para sus parámetros. Por ejemplo, en el siguiente fragmento de código vamos a invocar la función pre-definida llamada pow pasándole los valores 2 y 3 como *argumentos*.

>>> pow(2, 3)  
8

El fragmento nos muestra que el resultado de evaluar la función es el entero 8. El siguiente fragmento es similar al anterior, pero en este caso se inicia con una asignación: a la variable var se quiere dejar el valor de la expresión de la derecha. Como ya vimos, esto requiere que se *evalué* la parte derecha y, como en este caso tenemos una función, requiere la evaluación de la función con los parámetros dados. Al finalizar esta evaluación, en la variable var quedará el valor 8.

>>> var = pow(2, 3)  
>>> var  
8

Finalmente, el siguiente fragmento muestra dos ideas muy importantes sobre el uso y la evaluación de funciones. La primera, es que en Python es posible hacer composición de funciones: estamos llamando la función pow usando como segundo argumento el resultado de evaluar esa función usando los valores 2 y 2.

>>> var = pow(2, pow(2,1+1) )  
>>> var  
16

La segunda idea es que en Python se deben conocer los valores de los parámetros de una función *antes* de que se pueda evaluar la función. Esto significa que el intérprete Python realizará las siguientes acciones para ejecutar la primera instrucción del fragmento:

1. Evaluar la expresión 1+1. El resultado (2) se almacena temporalmente para usarse en el siguiente paso.
2. Evaluar la función pow con los argumentos 2 y 2. El resultado (4) se almacena temporalmente para usarse en el siguiente paso.
3. Evaluar la función pow con los argumentos 2 y 4. El resultado (16) se almacena temporalmente para usarse en el siguiente paso.
4. Asignar el valor 16 a la variable var.

Una anotación muy importante para hacer en este punto es que el orden de los parámetros es extremadamente importante y se debe respetar. Así como en el ejemplo *matemático* el valor de *f(2,3)* es diferente al valor de *f(3,2)*, en Python también pasa lo mismo. Por eso es absolutamente fundamental saber en qué orden fueron definidos los parámetros cuando se creó la función.

Si no se tiene acceso al código fuente, un mecanismo para consultar cuáles son los parámetros de una función es usar la función help. Esta es otra función básica de Python y permite consultar la documentación de cualquier otra función. Por ejemplo, en el siguiente fragmento se usará la función help para consultar la documentación de la función pow.

>>> help(pow)  
pow(x, y, z=None, /)  
 Equivalent to x\*\*y (with two arguments) or x\*\*y % z (with three arguments)  
  
 Some types, such as ints, are able to use a more efficient algorithm when  
 invoked using the three argument form.

### Funciones de conversión

En esta y las siguientes subsecciones describiremos varias funciones que son parte del lenguaje mismo y que sirven para resolver problemas que aparecen recurrentemente. Es una muy recomendable conocer estas funciones para poderlas utilizar cada vez que sea necesario, en lugar de estar repitiendo una y otra vez código que ya los responsables del lenguaje hicieron por nosotros.

La documentación completa y oficial de estas funciones se puede consultar en el siguiente link: <https://docs.python.org/3/library/functions.html>. Le recomendamos su consulta especialmente para que se familiarice con la estructura y el lenguaje de la documentación de Python.

Empezaremos discutiendo las 3 funciones para convertir entre tipos de datos que se presentaron brevemente en una sección anterior.

#### int(x)

La función[[10]](#footnote-10) int(x) convierte un valor x en un valor de tipo entero. La función int intenta ser tan versátil como sea posible y no requiere que x sea de un sólo tipo específico. Por eso es posible llamar a la función utilizando valores de tipo float o str, entre otros, y se puede suponer que la función aplicará la conversión adecuada para cada caso[[11]](#footnote-11).

* Si el parámetro x es de tipo float, el resultado de aplicarle la función int es la *parte entera* del valor. Es decir que el resultado de evaluar la expresión int(3.4) es 3, mientras que el resultado de evaluar la expresión int(-3.4) es -3. La función int *no hace* una aproximación matemática, sino que *trunca* los decimales.
* Si el parámetro x es de tipo str, el resultado de aplicarle la función int es el valor entero representado en la cadena de caracteres. Por ejemplo, si se evalúa la expresión int('-3') el valor resultante será el entero -3. En algunos casos, la función intentará hacer la conversión ajustando la cadena (por ejemplo si se evalúa la expresión int(' -3 ') que tiene espacios adicionales), pero fallará si la cadena no representa un número entero. Por ejemplo, los siguientes casos producirán un error:
  + int('3.2')
  + int(' - 3')
  + int('3+2')
* El error que se mostrará Python en estos casos será similar al siguiente:
* ValueError: invalid literal for int() with base 10: ' 3.2'
* Finalmente, si el parámetro x es de tipo int el resultado de aplicarle la función int es el mismo valor de x.

#### float(x)

La función float(x) convierte un valor x en un valor de tipo decimal. Esta función se puede llamar utilizando valores de tipo int o str, entre otros, y se puede suponer que la función aplicará la conversión adecuada para cada caso.

* Si el parámetro x es de tipo int, el resultado de aplicarle la función float es el mismo número, pero representado como un número decimal. Es decir que el resultado de evaluar la expresión float(3) es 3.0, y que el resultado de evaluar la expresión float(-3) es -3.0.
* Si el parámetro x es de tipo str, el resultado de aplicarle la función float es el valor decimal representado en la cadena de caracteres. Por ejemplo, si se evalúa la expresión float('-3.33') el valor resultante será el número -3.33. En algunos casos, la función intentará hacer la conversión ajustando la cadena (por ejemplo si se evalúa la expresión float(' -4 ') que tiene espacios adicionales y representa un entero), pero fallará si la cadena no representa un número. Por ejemplo, los siguientes casos producirán un error:
  + float('a')
  + float('3+2')
* El error que se mostrará Python en estos casos será similar al siguiente:
* ValueError: could not convert string to float: '3+2'
* Finalmente, si el parámetro x es de tipo float el resultado de aplicarle la función float es el mismo valor de x.

#### str(x)

La función str es una función tremendamente versátil que permite convertir un valor de cualquier tipo a una cadena de caracteres. Los siguientes son algunos ejemplos de invocaciones a la función str que se podrían hacer junto con el valor que calcularía la función.

* str(1), valor '1'
* str(-1.1), valor '-1.1'
* str(3.14), valor '3.14'
* str(1+2+0.14), valor '3.14'

### Funciones numéricas

Las 3 funciones anteriores sirven para hacer conversiones entre tipos de datos. A continuación, vamos a presentar unas funciones orientadas al trabajo con números.

#### abs(x)

La función abs(x) sirve para calcular el valor absoluto de un número. Esto quiere decir que si x es un número positivo, el resultado será el mismo x. En cambio, si x es un número negativo, el resultado será -x.

Si la función abs se invoca sobre algo que no sea un número, se producirá un error similar al siguiente:

TypeError: bad operand type for abs(): 'str'

#### round(n, digits)

La función round(n, digits) sirve para redondear un número para que sólo tenga la cantidad de decimales que nosotros le indiquemos. En este caso, el parámetro n hace referencia al número que queremos redondear, mientras que el parámetro digits especifica cuántos dígitos decimales queremos que tenga el resultado.

Los siguientes son algunos ejemplos del resultado de evaluar la función:

>>> round(3.14159, 0)  
3.0  
>>> round(3.14159, 1)  
3.1  
>>> round(3.14159, 2)  
3.14  
>>> round(3.14159, 3)  
3.142  
>>> round(3.14159, 4)  
3.1416  
>>> round(3.14159, 5)  
3.14159  
>>> round(3.14159, 6)  
3.14159

Es importante notar que, a diferencia de la función int, esta función sí hace una aproximación. Esto puede apreciarse en el siguiente ejemplo:

>>> int(2.7)  
2  
>>> round(2.7, 0)  
3.0

#### min, max

Las funciones min y max sirven para encontrar los valores mínimos y máximos de sus parámetros. El siguiente fragmento ilustra cuál sería el resultado de invocar estas funciones sobre 4 valores numéricos:

>>> min(3,2,6,7,5)  
2  
>>> max(3,2,6,7,5)  
7

#### pow(x, y)

La función pow se utiliza para calcular una potencia de un número. Es decir que pow(x, y) es equivalente a x \*\* y.

>>> pow(2,2)  
4  
>>> pow(2,3)  
8  
>>> pow(2,4)  
16

### Funciones de entrada y salida

A continuación, describiremos dos funciones que son tremendamente útiles para muchos programas y que son centrales para la interacción con el usuario. La primera función (print) servirá para mostrarle información a los usuarios, mientras que la segunda función (input) servirá para obtener información introducida por los usuarios del programa.

#### print

La función print ya la habíamos encontrado en secciones anteriores. Su objetivo es sencillamente imprimir información en la terminal o consola. Cuando se invoca la función, ella se encarga de imprimir el valor de cada uno de los parámetros separándolos con un espacio [[12]](#footnote-12).

El siguiente ejemplo además ilustra el hecho de que la cantidad de parámetros no es fija.

>>> print("Hola", "Mundo", "!")  
Hola Mundo !  
>>> print("Hola", "mundo", "!")  
Hola mundo !  
>>> print("Números", 123)  
Números 123  
>>> print("Números", 123, 456.0, "...")  
Números 123 456.0 ...

Un detalle importante de esta función es que permite combinar parámetros de diferentes tipos y todos los imprime. Para lograr esto, la función llama dentro de ella a la función str para cada uno de los parámetros.

#### input

La función input tiene como objetivo permitirle al usuario ingresar información para que el programa la utilice. Para esto la función primero le muestra al usuario un mensaje solicitándole la información y luego se queda esperando a que el usuario ingrese la información solicitada y presione la tecla *return*. Lo que haya tecleado el usuario se convierte en el valor de la invocación a la función.

El siguiente ejemplo ilustra el uso de la función:

>>> valor = input("Por favor ingrese el valor: ")  
Por favor ingrese el valor: 345  
>>> type(valor)  
<class 'str'>  
>>> valor  
'345'  
>>> numero = int(valor)  
>>> numero  
345

En la primera línea, vemos que la función se llamó usando un parámetro que le solicita al usuario que ingrese un valor. En esta línea también vemos que se está asignando a la variable llamada valor el resultado de evaluar la función input. Como ya se dijo, el valor de una invocación a esta función es igual al valor que ingrese el usuario.

En la segunda línea vemos que el programa imprimió el mensaje y vemos también el valor que tecleó el usuario (345 en este caso). El valor que el usuario tecleó queda almacenado en la variable valor. El valor almacenado **siempre** será una cadena de caracteres, como se ven en la siguiente línea cuando se revisa el tipo usando la función type. El valor de la variable es entonces la cadena '345' y por eso se debe hacer la conversión siguiente para poder usar el valor como un número.

### Funciones sobre caracteres

A continuación, vamos a describir dos funciones que, aunque no son utilizadas muy frecuentemente, resuelven problemáticas muy específicas e importantes.

#### Sistemas ASCII y UNICODE

Como el objetivo de esta sección no es hacer una descripción completa y pormenorizada de los sistemas ASCII y UNICODE, de su historia y de su uso para representar caracteres, vamos sólo a explicar lo mínimo necesario para entender esos mecanismos. El sistema ASCII se basa en una tabla que le asigna un número entre 0 y 255 a un conjunto de caracteres. Por ejemplo, la tabla dice que al caracter 'A' le corresponde el número 65, que al caracter 'a' le corresponde el 97, al caracter 'á' le corresponde el 192, al caracter '7' le corresponde el 55, y al caracter '!' le corresponde el 33.

Además de caracteres básicos (los que se podrían escribir con un teclado occidental), la tabla ASCII también incluye caracteres de control: como veremos, algunos son importantísimos mientras que otros están obsoletos y tenían sentido para cosas como controlar una impresora de punto.

El caracter de control más importante de todos es el que expresa un fin de línea y que en Python se representa como '\n'. En la tabla ASCII, este caracter tiene asignado el número 10. Otros caracteres de control importantes son el número 9 que representa una tabulación ('\t', el caracter que se introduce cuando se presiona la tecla TAB en un teclado) y el caracter 13 que representa un "retorno de carro" que en Python se representa usando '\r'. Aunque ya no tiene ninguna utilidad real, los computadores basados en el sistema operativo Windows aún representan el fin de línea usando la combinación de caracteres '\r\n' [[13]](#footnote-13).

Hoy en día no se utiliza propiamente la tabla ASCII en la mayoría de sistemas. Se usa en cambio UNICODE, que extiende ASCII y sirve para representar más de 137.000 caracteres incluyendo emoticons, kanjis y letras de alfabetos como el cirílico, árabe, hebreo o tailandés. El hecho de que sea una extensión significa que los primeros elementos de UNICODE son los mismos caracteres del sistema ASCII. Es decir que el caracter número 97 en UNICODE también es el caracter 'a'.

#### Funciones chr y ord

Después de haber presentado lo anterior, ahora sí podemos introducir las funciones chr y ord. La función chr permite consultar cuál caracter le corresponde a un número específico dentro del sistema UNICODE. Por ejemplo, si evaluamos la expresión chr(97) el resultado será una cadena de caracteres con el caracter 'a'. La función ord tiene el objetivo opuesto: dado un caracter, indica cual es el número que le corresponde en el sistema UNICODE.

Los siguientes son algunos ejemplos que permiten estudiar el uso de estas funciones. Dependiendo de la forma en la que esté visualizando este libro es posible que los ejemplos con caracteres especiales no aprecien correctamente.

>>> chr(98)  
'b'  
>>> ord('B')  
66  
>>> ord('☻')  
9787  
>>> chr(9787)  
'☻'  
>>> chr(22223)  
'囏'

Finalmente, si se invoca la función ord usando una cadena con más de un caracter se producirá el siguiente error:

TypeError: ord() expected a character, but string of length 3 found

## Ejercicios

1. ¿Qué función básica del lenguaje utilizaría para realizar las siguientes actividades?
   * Calcular el valor mínimo entre 5 números.
   * Convertir una cadena de caracteres a un número decimal.
   * Pedirle al usuario un número entero.
2. Consulte cuáles serían los números correspondientes a los siguientes caracteres dentro del sistema Unicode.
   * z
   * G
   * 0
   * 1
   * 2
   * $
   * \
3. Consulte cuáles serían los números correspondientes a los siguientes caracteres dentro del sistema Unicode:
   * El emoji para una cara triste.
   * El emoji para una bandera de Colombia.
   * El kanji japonés para representar la palabra árbol.
4. Escriba un programa que pida al usuario una cantidad de pesos, una tasa de interés y un número de años. Muestre por pantalla en cuánto se habrá convertido el capital inicial transcurridos esos años si cada año se aplica la tasa de interés introducida. Recuerde que un capital de C pesos a un interés del x por cien durante n años se convierten en C(1 + x /100)n pesos. (Pruebe su programa sabiendo que una cantidad de 10,000 pesos al 4.5 % de interés anual se convierte en 24,117.14 pesos al cabo de 20 años)
5. Escriba un programa que le pida al usuario 3 valores y los almacene en 3 variables enteras llamadas x1, x2 y x3. El programa luego debe rotar las variables de forma que al final x2 tenga el valor inicial de x1, x3 el de x2 y x1 el de x3.

## Más allá de Python

Esta sección se concentró en la definición de función que utiliza Python y en la descripción de funciones básicas que hacen parte de la librería fundamental del lenguaje. En otros lenguajes de programación el concepto de función es idéntico, mientras que en otros es mucho menos importante. Por ejemplo, en Java o en C++ el concepto que se usa para agrupar instrucciones, nombrarlas y poder invocarlas es el concepto de método. En Python existe también un concepto similar, pero es este libro no lo estudiaremos puesto que está atado al concepto de clase.

Cuando se cambia de lenguaje de programación, una dificultad importante tiene que ver con aprender el nombre y la forma de usar una buena cantidad de elementos básicos del lenguaje, similares a las funciones que estudiamos en esta sección. Por ejemplo, mientras que en Python se usa la función print para mostrar algo en la consola, en C tendría que usarse printf y en Java tendría que usarse el método println del objeto System.out.

Aunque conceptualmente no sean difícil entender las diferencias y empezar a usar estas otras funciones, aprender todos esos pequeños detalles hace un poco más largo el proceso de pasar de un lenguaje a otro.

En esta sección describimos también el mecanismo de evaluación de funciones, el cual es "eager", mientras que en otros lenguajes es "lazy" (perezoso): esto quiere decir que en esos lenguajes los valores de los parámetros se evalúan en el último momento posible, cuando realmente se necesite su valor. Esto no hace que Python sea malo o ineficiente, pero en algunas situaciones podría llevar a problemas con el desempeño de un programa.

# Definición de funciones

## Antes de empezar: archivos .py

Hasta el momento sólo hemos trabajado con pequeños programas que se pueden ejecutar y probar fácilmente en el REPL. A partir de este momento vamos a construir programas mucho más grandes que esperaríamos ejecutar varias veces.

En Python existe un concepto llamado *módulo*, que no es más que un archivo con definiciones de funciones e instrucciones adicionales. El nombre del archivo tiene que ser de la forma nombre\_modulo.py, y todas las funciones que se definan dentro del archivo se considerarán parte del módulo.

La forma más usual para ejecutar un módulo es utilizar la línea de comandos e invocar el programa Python con el nombre del archivo. Por ejemplo, para ejecutar el módulo definido en el archivo hola\_mundo.py es necesario usar la siguiente instrucción en la línea de comandos:

python hola\_mundo.py.

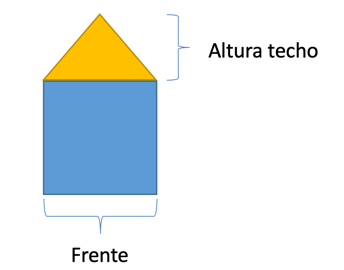
Los módulos también se pueden ejecutar desde los IDEs, usando la funcionalidad que cada uno tiene para ello. En el caso de Spyder, lo que se tiene que hacer es ubicar el cursor sobre el archivo que contiene nuestro módulo, y presionar la tecla **F9**: veremos entonces el resultado de la ejecución en la ventana de la consola. También es posible ejecutar un módulo haciendo clic en el botón con un triángulo verde que se encuentra en la parte superior.

Finalmente, veamos que ejecutar un programa es exactamente lo mismo que ejecutar un módulo. Lo único que se debe notar es que usualmente un programa involucra varios módulos y que para ejecutar el programa se debe saber cuál es el módulo que se debe usar para iniciar el programa.

## Un ejemplo completo

En la sección anterior definimos el concepto de función y lo ilustramos con varias funciones básicas del lenguaje Python. Como parte de esto, también se mostraron varios ejemplos de invocaciones a estas funciones y se presentaron las principales reglas para la evaluación de funciones. Esta sección completa la discusión sobre funciones en Python explicando cómo se pueden definir nuevas funciones.

Para empezar, presentamos un programa completo que al final de esta sección usted debería ser capaz de explicar y reconstruir. Léalo con cuidado, teniendo en que cuenta que cuando se habla de "una casa" se hace referencia a un dibujo como el siguiente:



# Este programa está en el archivo casa.py  
  
def area\_cuadrado(lado: int)-> int:  
 """  
 Calcula el área de un cuadrado dada la medida de su lado  
 """  
 return lado \* lado  
  
  
def area\_triangulo(base: int, altura: int)-> float:  
 """  
 Calcula el área de un triángulo dada la medida de la base y de la altura.  
 """   
 return (base \* altura) / 2  
  
  
def area\_casa(frente: int, techo: int)-> float:  
 """  
 Calcula el área del dibujo de una casa que se forma con un cuadrado  
 y un triángulo encima (el techo).  
 El frente de la casa será igual al lado del cuadrado y a la base del triángulo.  
 La altura del techo será la altura del triángulo.  
 """  
 cuadrado = area\_cuadrado(frente)  
 triangulo = area\_triangulo(frente, techo)  
 return cuadrado + triangulo  
   
medida\_frente = 7  
medida\_techo = 5  
resultado = area\_casa(medida\_frente, medida\_techo)  
print("El área de una casa con", medida\_frente, "metros de frente y un techo de",  
 medida\_techo, "metros de alto es ", round(resultado, 2), "metros")

Como siempre, no se preocupe si hay cosas en el programa anterior que no haya entendido: todo se irá aclarando a lo largo de la sección.

## Definición de funciones en Python

En la sección anterior vimos que Python incluye varias funciones que nosotros podemos utilizar aunque no sepamos cuáles sean exactamente las instrucciones que ejecuta cada una. Lo que conocemos nosotros de esas funciones es lo que llamamos *signatura*, la cual incluye el nombre, los parámetros que espera y su retorno. Lo que desconocemos es el *cuerpo* o *implementación* de las funciones, es decir las instrucciones que hacen que la función cumpla con lo que se espera de ella.

### La signatura de una función

La signatura de una función puede verse como la especificación de las reglas para utilizar la función y está compuesta por tres cosas:

1. **El nombre**. Este debería ser un nombre claro y fácil de recordar. Además, no debería estar repetido para que no haya ambigüedad cuando se quiera invocar a la función.
2. **Los parámetros**. Estos son los valores que se le tienen que pasar a la función cuando se quiera invocar. Pueden verse como la información que tiene que proporcionar quien llame a la función para que se pueda cumplir con su objetivo. Una función puede tener uno o varios parámetros.
3. **El resultado**. En tercer lugar, tenemos información sobre el resultado de la función que nos dice si será un número, una cadena de caracteres o cualquier otra cosa.

Volvamos ahora al ejemplo para identificar estos elementos:

def area\_cuadrado(lado: int)-> int:  
 """  
 Calcula el área de un cuadrado dada la medida de su lado  
 """  
 return lado \* lado

En este ejemplo, se está definiendo una función y se ha incluido tanto la signatura como el cuerpo. Por ahora vamos a analizar sólo la signatura, que en este caso corresponde a la primera línea del ejemplo.

Lo primero que nos encontramos es la palabra reservada[[14]](#footnote-14) de Python def que nos marca el inicio de la definición de una función. La signatura de la función va hasta los dos puntos (:) que se encuentran al final de la línea.

Después de la palabra def, viene el nombre que le queremos dar a la función. En este caso escogimos area\_cuadrado para expresar claramente su objetivo: será una función para calcular el área de un cuadrado. Como los nombres de las funciones no pueden tener espacios, hemos separado las dos palabras usando el símbolo \_[[15]](#footnote-15).

El siguiente punto es la definición de los parámetros de la función, así que deberíamos hacernos la pregunta: ¿si queremos calcular el área de un cuadrado, qué información requerimos para poder hacer el cálculo? En este caso la respuesta es que sólo necesitamos la medida del lado del cuadrado. Es decir que necesitamos un parámetro en la función y que ese parámetro debería representar la longitud del lado del cuadrado.

En nuestro ejemplo, el parámetro está especificado en la parte que dice lado: int. Esto significa que la función esperará un solo parámetro, que nos vamos a referir a ese parámetro como lado, y que ese parámetro debería ser un número entero (int). Los parámetros siempre se especifican dentro de un par de paréntesis.

Finalmente encontramos la especificación del resultado de la función. En nuestro ejemplo, la signatura especifica que su resultado será un número entero con el fragmento que dice -> int.

En resumen: la signatura de una función especifica cómo se debería invocar la funcón y qué se debería esperar como resultado. En nuestro caso de ejemplo, la función se invocará con el nombre area\_cuadrado, requerirá que se use un parámetro de tipo int y generará como resultado otro número entero (int).

#### Un segundo ejemplo

Analicemos ahora la segunda función de nuestro ejemplo:

def area\_triangulo(base: int, altura: int)-> float:  
 """  
 Calcula el área de un triángulo dada la medida de la base y de la altura.  
 """   
 return (base \* altura) / 2

¿Qué nos dice la signatura con respecto a la función?

* ¿Cómo se llama la función?
* ¿Cuántos parámetros se requieren para su invocación?
* ¿De qué tipo son los parámetros?
* ¿De qué tipo será el resultado de la función?

Lo único diferente con respecto al primer ejemplo es que tenemos dos parámetros en lugar de uno y que esos dos parámetros se separaron utilizando una coma.

**Actividades:**

1. Escriba la signatura de una función que sirva para calcular el área de un círculo dado su radio.
2. Escriba la signatura de una función que sirva para calcular la velocidad final de un objeto que tiene una velocidad inicial y acelera a una tasa constante durante una cierta cantidad de tiempo.

### El cuerpo de una función

Pasamos ahora a estudiar el cuerpo de la primera función y lo primero que debemos notar es que todo el cuerpo está **indentado**. Esto quiere decir, que todo lo que hace parte del cuerpo de la función está escrito con un margen hacia la derecha. En este ejemplo particular el margen se ha creado usando 4 caracteres en blanco, lo cual corresponde a las buenas prácticas recomendadas de la comunidad Python.

def area\_cuadrado(lado: int)-> int:  
 """  
 Calcula el área de un cuadrado dada la medida de su lado  
 """  
 return lado \* lado

**Cuidado**: La indentación no es opcional en Python, sino obligatoria. El cuerpo de todas las funciones tiene que estar indentado y el margen utilizado debe ser consistente: si una línea usa un cierto margen y la siguiente tiene más o menos caracteres, se producirá un error. La recomendación en este libro será siempre utilizar 4 caracteres en blanco para la indentación.

Las primeras líneas del cuerpo de esta función nos muestran otra forma de introducir comentarios en un programa, esta vez asociados a una función. En Python, si la primera línea del cuerpo de una función inicia con una cadena de caracteres, esa cadena se convertirá en la documentación asociada a la función y aparecerá cuando alguien llame a la función help usando el nombre de nuestra función.

En general, siempre debería incluirse documentación en las funciones, por más sencillas y evidentes que sean. En este ejemplo se ha incluido sólo una documentación breve para la función area\_cuadrado, pero para casos más complejos habría sido conveniente documentar también los parámetros y el retorno de la función. Más adelante estudiaremos algunas buenas prácticas para completar la documentación de las funciones.

Además de la documentación, el cuerpo de la función area\_cuadrado sólo tiene una instrucción:

return lado \* lado

La interpretación de esta instrucción es muy sencilla: cuando se ejecute esta instrucción, la función deberá *retornar* el valor de la expresión lado \* lado. En el contexto de la ejecución de una función, retornar hace referencia a responderle con un valor a quien haya invocado la función. Esto quiere decir que cuando se llega a una instrucción return, la ejecución de la función termina y se responde con un valor de acuerdo con lo especificado en la signatura de la función.

Ahora bien, antes de retornar un valor, es necesario que Python evalué la expresión lado \* lado y encuentre a qué valor equivale. Si no tuviéramos contexto, lo que podríamos suponer es que lado fuera el nombre de una variable y que la expresión está calculando el cuadrado de la variable. En realidad, no hemos definido ninguna variable con ese nombre, pero sí tenemos un *parámetro* con ese nombre en la signatura de la función. La expresión lado \* lado está haciendo entonces referencia al parámetro que le pasen a la función cada vez que la invoquen.

Viendo todo en conjunto, lo que pasará cuando alguien invoque a nuestra función es lo siguiente:

1. El que invoque la función, tendrá que darle un valor al parámetro lado. La invocación podría ser algo como area\_cuadrado(4).
2. Nuestra función intentará ejecutar la instrucción return lado \* lado. Como la parte derecha es una expresión cuyo valor se desconoce, se tendrá que evaluar primero.
3. Para evaluar la expresión lado \* lado se debe conocer el valor de lado. Como es un parámetro, entonces se tomará el valor que se le haya dado cuando se invocó a la función. En este caso, la expresión se convertiría en 4 \* 4.
4. Como la expresión sigue sin tener un valor, se ejecuta la operación de multiplicación y la expresión se convierte en el valor 16.
5. Se ejecutará la instrucción return 16: terminará la ejecución de la función y el valor 16 se le pasará al que haya invocado la función.

#### Un segundo ejemplo

Analicemos ahora la segunda función de nuestro ejemplo:

def area\_triangulo(base: int, altura: int)-> float:  
 """  
 Calcula el área de un triángulo dada la medida de la base y de la altura.  
 """   
 return (base \* altura) / 2

* ¿Qué hace el cuerpo de esta función?
* ¿Qué diferencias tiene con respecto al cuerpo de la primera función?

**Actividades:**

1. Escriba el cuerpo de la función para calcular el área de un círculo dado su radio. Utilice la signatura que definió en la actividad anterior.
2. Escriba el cuerpo de la función para calcular la velocidad final de un objeto. Utilice la signatura que definió en la actividad anterior.

## Definir vs. Invocar

Hasta el momento hemos descrito en detalle solamente la forma en la que se define una función, pero no hemos estudiado cómo se invoca una función. Es muy importante tener muy clara la diferencia entre estas dos acciones: cuando definimos una función, sólo le estamos *enseñando* a Python cómo debería invocarse esa función y cómo debería comportarse un programa cuando la función se invoque; cuando invocamos una función, le estamos pidiendo a Python que *ejecute* el cuerpo de la función, dándole valores específicos a cada uno de sus parámetros.

Revisaremos a continuación la última parte del programa de ejemplo con el que inició esta sección. Para simplificar el ejemplo hemos eliminado la documentación de la función area\_casa.

def area\_casa(frente: int, techo: int)-> float:  
 cuadrado = area\_cuadrado(frente)  
 triangulo = area\_triangulo(frente, techo)  
 return cuadrado + triangulo  
   
   
medida\_frente = 7  
medida\_techo = 5  
resultado = area\_casa(medida\_frente, medida\_techo)  
print("El área de una casa con", medida\_frente, "metros de frente y un techo de",  
 medida\_techo, "metros de alto es ", round(resultado, 2), "metros")

### Definición de la función

Lo primero que se debe notar en este ejemplo es que las primeras 4 líneas hacen parte de la definición de la función area\_casa mientras que el último bloque de instrucciones está por fuera de esta función. Esto es evidente gracias a la indentación: las instrucciones indentadas después de la signatura de la función, hacen parte del cuerpo de la función. Tan pronto encontramos instrucciones que no están indentadas significa que hemos encontrado el fin del cuerpo de la función.

Con respecto a las funciones anteriormente estudiadas, la signatura de la función area\_casa sólo tiene una pequeña diferencia: utiliza diferentes tipos de datos (sus parámetros son int y su resultado es float).

En cambio, el cuerpo de la función tiene un par de diferencias mucho más interesantes. La primera es que dentro de esta función se definen dos variables (cuadrado y triangulo). Estas variables son *locales*, lo cual significa que sólo existirán dentro del contexto de una invocación a la función. Si se habla de esas variables por fuera del cuerpo de la función, se producirá un error porque esos nombres sólo están definidos dentro del *alcance* de la función area\_casa.

La segunda diferencia interesante tiene que ver con la forma en la que se le asignan valores a las nuevas variables. En el caso de la variable cuadrado, se le está asignando el valor obtenido al invocar la función area\_cuadrado utilizando como argumento el valor del parámetro frente. En otras palabras, cuando se quiere calcular el área de una casa dada la medida de su frente y la altura de su techo, lo primero que se hace es calcular el área del cuadrado utilizando para eso la función area\_cuadrado, dándole a su parámetro lado el valor que se le haya dado al parámetro frente.

La segunda instrucción hace algo similar para darle un valor a la variable triangulo. En este caso, la invocación se hace a la función area\_triangulo y se utilizan como argumento los valores de los parámetros frente y techo.

La tercera instrucción de la función calcula la suma de las dos variables y retorna el resultado.

El nombre que se le da a un parámetro en la signatura de una función sólo es importante dentro del cuerpo de esa función. En el caso del ejemplo, la función area\_cuadrado tiene un parámetro llamado lado, pero quien invoca a esa función no tiene por qué usar el mismo nombre para sus propios parámetros o variables.

### Invocación de la función

Si nuestro programa tuviera sólo las instrucciones que ya estudiamos, no veríamos nada pasando cada vez que lo corriéramos. Su ejecución se limitaría a definir las funciones una y otra vez, pero sin invocarlas ni una sola vez. En este contexto es que se vuelven interesantes las últimas instrucciones del archivo, las cuales repetimos a continuación:

medida\_frente = 7  
medida\_techo = 5  
resultado = area\_casa(medida\_frente, medida\_techo)  
print("El área de una casa con", medida\_frente, "metros de frente y un techo de",  
 medida\_techo, "metros de alto es ", round(resultado, 2), "metros")

Recuerde que estas instrucciones no hacen parte de la definición de ninguna función, así que se ejecutarán cada vez que el programa se corra. Las dos primeras instrucciones hacen asignaciones sobre dos nuevas variables llamadas medida\_frente y medida\_techo con los valores 7 y 5.

A continuación, se hace una nueva asignación pero esta vez el valor se calcula con una invocación a la función area\_casa. En esta ocasión la función sí se ejecuta, utilizando los valores 7 y 5 como argumentos de la invocación. Si nosotros no conociéramos el cuerpo de la función, lo único que podríamos ver es que el resultado de invocar la función quedaría asignado a la variable resultado.

En este caso, como sí conocemos el cuerpo de la función, sabemos que lo que ocurrirá es lo siguiente:

1. Se invocará a la función area\_cuadrado usando el valor 7 como valor para el parámetro lado.
2. Se calculará y retornará el valor de la expresión lado \* lado que se encuentran en el cuerpo de area\_cuadrado. En este caso, el valor retornado será 49 y ese valor se almacenará en la variable temporal cuadrado.
3. Se invocará a la función area\_triangulo usando los valores 7 y 5 como valores para los parámetros base y altura.
4. Se calculará y retornará el valor de la expresión (base \* altura) / 2 que se encuentran en el cuerpo de area\_triangulo. En este caso, el valor retornado será 17.5 y ese valor se almacenará en la variable temporal triangulo.
5. Se calculará la suma de cuadrado y triangulo y se retornará el valor. Las variables cuadrado y triangulo dejan de existir en este momento porque terminó la ejecución de la función en la que fueron definidas.
6. El valor retornado se almacenará en la variable resultado.

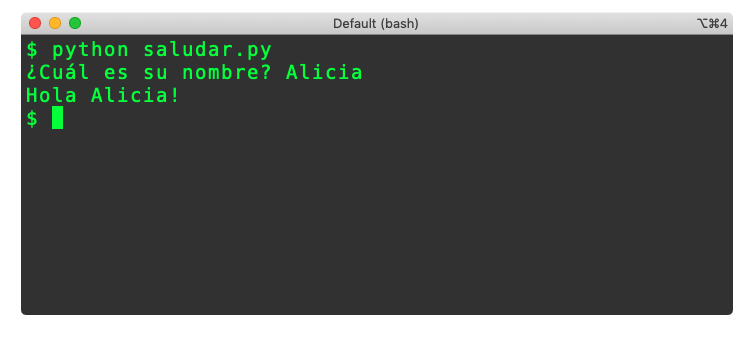
Finalmente se debe ejecutar la última instrucción del programa, que en este caso es una invocación a la función print. Como ya sabemos, esta función le mostrará al usuario los valores que se le pasen como parámetro, separándolos con un espacio. Sin embargo, los argumentos que se están usando para llamar a la función print incluyen la expresión round(resultado, 2). Esto quiere decir que antes de que se empiece a ejecutar la función print se llamará a la función round y se obtendrá un valor redondeado para la variable resultado. Esto es un ejemplo del uso de invocaciones a funciones como argumentos de una invocación a otra función.

### Ejecución de un programa

Volvamos ahora a la discusión del inicio de la sección sobre cómo se ejecuta un módulo y revisemos lo que pasa en cada paso. Para eso, tomemos como ejemplo el siguiente programa, que suponemos que está escrito en el archivo 'saludar.py':

def saludar(nombre: str)-> str:  
 return "Hola " + nombre + "!"  
  
nombre = input("¿Cuál es su nombre? ")  
saludo = saludar(nombre)  
print(saludo)

La siguiente imagen muestra cómo se ejecutó el programa y el resultado de la ejecución:



Ahora analicemos lo que ocurrió en cada paso de la ejecución:

1. Al invocar al programa 'python' usando como parámetro el nombre de archivo 'saludar.py', el archivo se abre y Python empieza a revisarlo bloque por bloque e instrucción por instrucción.
2. En primer lugar, se encuentra con la definición de la función saludar. Python revisa que la signatura esté definida con la sintaxis correcta. Por ejemplo, si el parámetro dijera que es de tipo srr en lugar de str, habría aparecido un error diciendo NameError: name 'srr' is not defined.
3. Ahora Python revisa todas las instrucciones que se encuentran en el cuerpo de la función, que en este caso es solamente una. La revisión es, nuevamente, sintáctica (sólo está revisando que el código esté bien escrito, no que sea correcto).
4. Como terminó la definición de la función, Python almacena la definición en un registro por si más adelante alguien invoca a una función con ese nombre y ese número de parámetros. Note que hasta este punto la función **no se ha ejecutado**.
5. Se revisa ahora lo siguiente en el archivo, que es una instrucción de asignación. Como sabemos, lo primero que se hace es evaluar la parte de la derecha que en este caso requiere invocar la función input. En la captura de pantalla vemos que el mensaje se imprimió para preguntarle al usuario por su nombre y que el usuario respondió con su nombre ('Alicia'). Finalmente, esta cadena queda almacenada en la variable nombre.
6. La siguiente instrucción también es una asignación, así que se evalúa la parte derecha primero. En este caso se debe invocar la función saludar, usando como parámetro el valor almacenado en la variable nombre: es ahora cuando se ejecuta por primera vez nuestra función saludar.
7. La siguiente instrucción que se ejecuta es la instrucción que se encuentra dentro del cuerpo de la función. Lo primero que hace esta función es calcular el valor de la expresión "Hola " + nombre + "!". En el caso del ejemplo, el valor del parámetro nombre es la cadena 'Alicia', así que el valor de la expresión completa es la cadena 'Hola Alicia!. Finalmente, se retorna esta cadena completa y termina la ejecución de la función.
8. De vuelta al programa principal, el resultado de la función se almacena en la variable saludo.
9. Por último, se invoca la función print y se le pasa como argumento el valor contenido en la variable saludo. El programa imprime entonces 'Hola Alicia! tal como se ve en la captura de pantalla.

El siguiente es nuevamente el código de nuestro programa, pero esta vez hemos incluido unos comentarios adicionales que indican el orden en el que se van *ejecutando* las instrucciones.

def saludar(nombre: str)-> str: #1  
 return "Hola " + nombre + "!" #4  
  
nombre = input("¿Cuál es su nombre? ") #2  
saludo = saludar(nombre) #3 #5  
print(saludo) #6

**Actividades:**

1. Cree el módulo "cuadrados.py" y defina una función que calcula el perímetro de un cuadrado y otra que calcula su área. Agregue al archivo las instrucciones para preguntarle al usuario por el lado de un cuadrado y luego mostrarle el perímetro y el área de un cuadrado con esa medida.

## Funciones sin parámetro o sin retorno

Dos preguntas frecuentes entre los estudiantes son si una función siempre debe tener parámetros y si una función siempre debe retornar un valor. Si estas preguntas se hicieran en un contexto matemático, la respuesta sería negativa: las funciones establecen relaciones entre elementos de un conjunto y elementos de otro conjunto, así que siempre tienen al menos un parámetro y siempre tienen un resultado.

En el contexto de Python, sí es posible tener funciones sin parámetros y funciones que no tengan un retorno, pero la realidad es que este tipo de funciones sólo deberían usarse en un contexto muy particular (interacción con el usuario). A continuación, explicamos brevemente por qué, en general, no es una buena idea tener este tipo de funciones.

### Funciones sin parámetros

Si una función no tiene parámetros, en principio siempre va a retornar el mismo valor. Si esto fuera cierto, en lugar de tener una función para calcular el valor, se debería tener una variable donde ese valor estuviera disponible para utilizarlo cuando fuera necesario.

Python incluye algo así para almacenar el valor de Pi sin que sea necesario llamar cada vez a una función que lo calcule.

Hay 3 casos relativamente sencillos en los cuales tendría sentido tener funciones sin parámetros:

1. Cuando los datos para calcular el resultado de la función dependan de valores entregados por el usuario. En ese caso, aunque no se vería en la signatura, la función recibiría valores que cambiarían su resultado en cada ejecución.
2. Cuando el resultado de la función pueda cambiar con el paso del tiempo. Por ejemplo, una función que calculara un valor aleatorio podría no requerir ningún parámetro para cambiar el valor que retorne cada vez.
3. Cuando la función dependa de valores externos a ella, pero que no tengan que llegar por parámetro. Por ejemplo, una función podría depender de una variable global o de una condición del ambiente de ejecución, como la hora o la ruta desde la que se esté corriendo el programa.

### Funciones sin retorno

Si una función no tiene un retorno significa que el resultado de ejecutar sus instrucciones no es interesante para el resto del programa, o es interesante sólo por los efectos que haya podido tener en algún otro elemento del programa o del computador. Por ejemplo, una función sin retorno podría utilizarse para eliminar un archivo: la función eliminaría el archivo y no retornaría ningún valor.

En general, las funciones podrían no tener retorno cuando representen acciones que quieren realizarse y no tengan ningún resultado que informar.

## Sobre los *type-hints*

Si usted utiliza otros libros o si consulta en Internet, es muy posible que se encuentre con definiciones de funciones en las que no aparecen los tipos de los parámetros ni el tipo de los resultados. Esto se debe a que en Python el uso de estos elementos es opcional. De hecho, el nombre específico de estos elementos es *type-hints* (sugerencias de tipo) y las herramientas (IDE, intérprete, compilador, etc.) los utilizan sólo como sugerencias.

En este libro vamos a usar *type-hints* en la definición de todas las funciones y esperamos que usted haga uso de ellos también. Por una parte, esto le facilitará aprender a usar otros lenguajes de programación como C, C++, Java, o TypeScript. Por otro lado, razonar sobre los tipos de datos debería ayudarlo a estructurar mejor sus programas, especialmente mientras adquiere una cierta destreza programando.

## Ejercicios

1. Defina una función que permita convertir de grados Celsius a grados Fahrenheit.
2. Defina una función que permita convertir de grados Fahrenheit a grados Celsius.
3. Escriba un programa que le pida al usuario una temperatura en grados Celsius y le informe a cuánto equivaldría esa temperatura en grados Fahrenheit.
4. Escriba un programa que le pida a un usuario una cantidad de días y le muestre la cantidad de años, meses y días equivalentes. Suponga que todos los meses tienen 30 días.

## Más allá de Python

En esta sección utilizamos una definición de función relativamente *pura*, en la cual el resultado de su ejecución depende únicamente del valor que se les dé a sus parámetros. En otros contextos, y especialmente en otros lenguajes de programación, la ejecución de funciones, métodos o procedimientos depende de otros factores que no se ven explícitos en sus signaturas. Por ejemplo, en un lenguaje orientado a objetos como Java o C++, un método definido en una clase depende de los parámetros y del estado del objeto sobre el que se invoque el objeto. Esto hace mucho más complejo el comportamiento de un método y hace que ciertas acciones, como probar su corrección, sea más difícil que en funciones equivalente en Python.

La discusión sobre los *type-hints* tiene que ver con una discusión mucho más extensa sobre la conveniencia de tener *tipado dinámico* en los lenguajes de programación. Por un lado, cuando los lenguajes son fuertemente tipados se cometen menos errores o, al menos, las herramientas de edición capturan más errores de forma temprana. Por otro lado, cuando el tipado es dinámico los errores de tipo se capturan en tiempo de ejecución, pero el desarrollo de los programas es más rápido. En este momento hay fuertes discusiones sobre la conveniencia o no de cada sistema, pero hay un hecho que encontramos muy diciente: JavaScript, que tiene tipado dinámico, está incluyendo progresivamente más elementos para escribir programas fuertemente tipados, mientras que Python está empezando a incluir elementos para poder incluir verificaciones de tipos.

Python utiliza un estilo para el nombramiento de variables, funciones y parámetros que está descrito en la guía PEP8 (Style Guide for Python Code), bajo el título "Naming Conventions". El punto que vale más la pena resaltar es el que tiene que ver con la separación de palabras en un identificador: mientras que en Python podemos encontrar una función llamada area\_casa, en Java una función similar se llamaría areaCasa y en C# se llamaría AreaCasa. Estas diferencias son en realidad minúsculas, pero es muy recomendable seguir las guías de estilo de la plataforma en la que se esté trabajando, para facilitar la lectura y evitar incompatibilidades.

La guía de estilo oficial de Python, que todos los programadores deberían conocer e intentar aplicar, se encuentra en el siguiente vínculo: PEP8 -- Style Guide for Python Code: <https://www.python.org/dev/peps/pep-0008/>

# Un programa para leer (2)

Volvemos ahora al programa completo que usted leyó al inicio del capítulo. Vuelva a leerlo detenidamente: todo lo que aparece en este programa ya lo estudiamos y usted debería estar en capacidad de entenderlo.

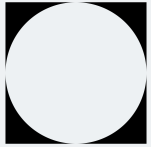
# Este programa está escrito en el archivo perimetro.py  
  
def perimetro\_triangulo(cateto1: float, cateto2: float)->float:  
 """  
 Esta función calcula el perímetro de un triángulo rectángulo  
 dada la longitud de sus dos catetos  
 Parámetros:  
 cateto1 (float): la longitud del primer cateto del triángulo  
 cateto2 (float): la longitud del segundo cateto del triángulo  
 Retorno  
 (float): la longitud del perímetro del triángulo  
 """  
 # Usar la función calcular\_hip para calcular la longitud del lado faltante  
 hipotenusa = calcular\_hip(cateto1, cateto2)  
   
 # Sumar los tres lados y convertirlos en la respuesta de la función  
 return cateto1 + cateto2 + hipotenusa  
  
  
def calcular\_hip(cateto1: float, cateto2: float)->float:  
 """  
 Esta función calcula la longitud de la hipotenusa en un triángulo rectángulo  
 dada la longitud de sus dos catetos  
 Parámetros:  
 cateto1 (float): la longitud del primer cateto del triángulo  
 cateto2 (float): la longitud del segundo cateto del triángulo  
 Retorno  
 (float): la longitud de la hipotenusa  
 """  
 # Sumar la longitud de los catetos elevados al cuadrado  
 suma\_cuadrados = (cateto1 \*\* 2) + (cateto2 \*\* 2)  
   
 # Calcular la raiz cuadrada de la suma usando la función pow y el exponente 0.5  
 hipotenusa = pow(suma\_cuadrados, 0.5)  
 return hipotenusa  
  
  
# Solicitarle al usuario la longitud de los dos catetos  
cadena\_cat\_1 = input("Indique la longitud del primer cateto: ")  
cadena\_cat\_2 = input("Indique la longitud del segundo cateto: ")  
  
# Convertir los caracteres dados por el usuario en un número decimal  
cat\_1 = float(cadena\_cat\_1)  
cat\_2 = float(cadena\_cat\_2)  
  
# Llamar a la función con los valores recibidos  
perimetro = perimetro\_triangulo(3,4)  
  
# Mostrarle al usuario el resultado de la operación  
print("El perímetro de un triángulo rectángulo que tenga catetos de longitud", cat\_1, "y", cat\_2, "es", perimetro)

**Preguntas:** A partir de su lectura del programa, intente responder las siguientes preguntas.

* ¿Cuál es el objetivo del programa?
* ¿Qué información tendrá que suministrar el usuario que ejecute el programa?
* ¿Cuál es el objetivo de cada bloque?
* ¿Qué es lo que primero se ejecuta?
* ¿Cual es la diferencia entre las cosas que están escritas en español y las que están escritas en inglés?
* ¿Cuáles son los valores que tiene que proporcionar el usuario?
* ¿Qué ve el usuario al finalizar la ejecución?

## Ejercicios

1. Copie el programa a su computador y ejecútelo. Hágale modificaciones y observe los cambios que se producen en sus resultados.
2. Escriba un programa completo que le pida al usuario la temperatura actual en grados Celsius y le informe cuál es esa temperatura en Fahrenheit.
3. Observe la siguiente figura:



Ejercicio Área

Construya un programa que le pida al usuario la medida del lado del cuadrado y le informe al usuario el tamaño del área de la zona negra.

# Lógica vs. Interacción

Usualmente la preocupación más importante cuando se construye un programa es que sea correcto, es decir que sus cálculos y la forma en la que manipule la información que le dé un usuario sea correcta. Por ejemplo, lo más importante del sistema que manipula el dinero en un banco es que lleve correctamente las cuentas: nadie puede perder dinero, deben cobrar lo correcto por cada operación, el descuento de impuestos debe ajustarse a la ley y deben abonar los intereses correctamente calculados.

Además de la corrección, hay otras preocupaciones para tener en cuenta cuando se desarrolla un nuevo programa. Para un usuario, las preocupaciones comunes tienen que ver con la seguridad, el desempeño (qué tan rápido es el sistema) y la usabilidad (qué tan fácil de usar es). Desde el punto de vista de quien desarrolla un sistema, otras preocupaciones importantes son la escalabilidad (qué tan bien funcionará el sistema cuando tenga muchos usuarios o muchos datos), la tolerancia a fallos (cómo se comporta el sistema cuando ocurren ciertos problemas), o la interoperabilidad (qué tan fácil es que el sistema intercambie información con otros sistemas que ya existan) [[16]](#footnote-16).

Sin embargo, la mayor preocupación para quien desarrolla un sistema debería ser la *mantenibilidad*. Esto quiere decir: qué tan fácil es hacerle un cambio a un sistema (ej. agregarle alguna funcionalidad, corregir algún error, mejorar su apariencia). Cuando se está aprendiendo a programar puede parecer que esta no es una cuestión importante y que lo prioritario son la corrección y el desempeño. La realidad es que una vez se empieza a usar un programa, siempre es necesario hacerle modificaciones, especialmente cuando es muy exitoso. Sólo piense en la cantidad de actualizaciones que tiene que instalar permanentemente en su computador o en su celular: con casi total seguridad usted no está utilizando la primera versión (1.0) de ninguna aplicación [[17]](#footnote-17).

La pregunta natural que debería surgir es entonces: ¿cómo logramos que un programa sea mantenible? Ya en este capítulo discutimos la primera recomendación: documentando el código. Cuando el código está documentado es más fácil que otros programadores o nosotros mismos podamos hacer mejoras o correcciones a nuestros programas.

La segunda recomendación es incluso más importante: estructurando el programa de tal forma que sea fácil de entender. Esto significa que queremos que sea fácil entender cómo está organizado para que cuando haya un error sea fácil encontrar donde se debería corregir, o para que cuando se quiera agregar nuevas funcionalidades no se necesite una gran reestructuración.

Es posible que en este momento sea difícil para usted imaginarse la complejidad de un gran desarrollo porque hasta ahora sólo ha trabajado con pequeños programas. Con el tiempo se dará cuenta que las recomendaciones para organizar sus programas son tremendamente valiosas y que le ahorrarán mucho trabajo en el futuro. **Acostúmbrese a estructurar bien sus programas mientras sean pequeños, para que sea natural hacerlo cuando sean grandes.**

En esta sección estudiaremos una primera técnica para propiciar la mantenibilidad de los programas. Incidentalmente, para explicar esta técnica tendremos que introducir el concepto de *módulo* en Python.

## Separación de la lógica y la interfaz

Como hasta el momento hemos construido programas relativamente pequeños, hemos usado un solo archivo para cada uno. A partir de este momento vamos a empezar a separar programa en dos módulos: uno para manejar la interacción con el usuario (pedirle datos, mostrarle información, etc.) y otro para manejar todo lo que consideramos la lógica del programa, es decir las instrucciones que son realmente el centro conceptual del programa.

Consideremos a modo de ejemplo, un programa que sirva para procesar información de un censo de población. El módulo con la interfaz de este programa servirá para que el usuario seleccione qué información quiere consultar y le mostrará las gráficas y tablas correspondientes. Por otro lado, la lógica del programa se encargará de calcular estadísticas, procesar los archivos y generar (¡no visualizar!) las gráficas que requiera el usuario.

Hay múltiples razones por los cuales esta separación es adecuada para programas como los que vamos a construir en este curso. La primera tiene que ver con el tipo de cosas que se hace en cada parte: mientras que en la interfaz todas las acciones deberían estar orientadas a interactuar con el usuario, en la lógica del programa hacemos cosas mucho más variadas. Con esto logramos que en cada módulo todo lo que hacemos esté relacionado: en lugar de tener funciones que le pidan información al usuario, hagan cálculos complejos y luego muestren el resultado, podemos tener en un módulo funciones dedicadas a la interacción y en otro módulo funciones dedicadas a hacer los cálculos.

Esta coherencia (o cohesión) termina llevando a programas que son mucho más fáciles de mantener porque dependiendo del tipo de error que se presente sabremos mejor dónde se debe corregir el código. La coherencia también se puede ver en el tipo de funciones que se utilizan: mientras en una interfaz basada en consola encontraremos muchos llamados a las funciones print e input, dentro de la lógica del programa encontraremos llamados muy diferentes que dependerán del problema que estemos resolviendo.

Otra ventaja de utilizar esta separación es que se vuelve más fácil probar la lógica de nuestros programas. Cuando la interfaz se mezcla con la lógica, para probar la corrección de un cálculo es necesario probar también la interfaz. Esto implica teclear datos cada vez y observar los resultados para compararlos con los esperados. Si la interfaz está separada de la lógica, podemos invocar directamente las funciones que hacen los cálculos, usando los mismos argumentos cada vez, es decir sin requerir la interacción con el usuario.

Otro motivo para hacer utilizar esta separación tiene que ver con la reutilización. Veremos más adelante que las interfaces de muchos programas son muy similares y que hay mucho código que fácilmente se puede adaptar para utilizar en un nuevo programa. Si la interfaz estuviera mezclada con la lógica, esta reutilización sería mucho más difícil.

## Implementación de módulos separados

Veamos ahora cómo se implementa en Python la separación entre la interfaz y la lógica de los programas. Para esto, tomaremos el programa 'saludar.py' que ya conocemos de una sección anterior:

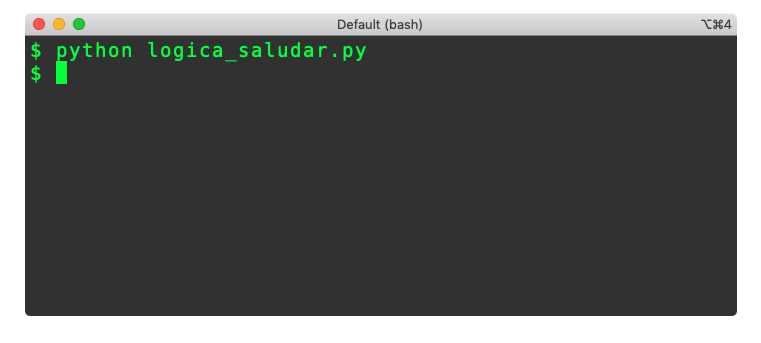
def saludar(nombre: str)-> str:  
 return "Hola " + nombre + "!"  
  
nombre = input("¿Cuál es su nombre? ")  
saludo = saludar(nombre)  
print(saludo)

Empezaremos primero con la creación del archivo con la lógica, al cual llamaremos 'logica\_saludar.py'.

# Esto está en el archivo logica\_saludar.py  
  
def saludar(nombre: str)-> str:  
 return "Hola " + nombre + "!"

Como se puede ver, en este archivo únicamente dejamos la función que hace los "cálculos" del programa y crea un mensaje para saludar. Aunque el ejemplo es extremadamente sencillo, es importante notar que en este archivo no se está usando ni la función print ni la función input.

Otra cosa importante para analizar es lo que pasaría si decidiéramos ejecutar este módulo usando la siguiente instrucción en la línea de comandos: python logica\_saludar.py.



Como se ve en la captura de pantalla, no hay ninguna evidencia en la consola de lo que sucedió y parecería que nada hubiera pasado. En realidad, lo que hizo el intérprete de Python fue lo siguiente:

1. Abrió el archivo
2. Leyó la signatura de la función saludar y revisó que fuera sintácticamente correcta.
3. Revisó el cuerpo de la función saludar y revisó que fuera sintácticamente correcto.
4. Como no había errores, agregó la función a la lista de funciones que se podrían invocar.
5. Terminó la ejecución.

Es decir que, aunque se leyó la definición de la función, no hubo ninguna invocación ni se ejecutó ninguna instrucción.

Pasemos ahora al módulo en el que vamos a implementar la interfaz de la aplicación. El archivo que usaremos se llamará 'interfaz\_saludar.py' y estará ubicado en *la misma carpeta* que el archivo 'logica\_saludar.py'. Si los archivos están en carpetas diferentes, no funcionará todo lo que explicaremos a continuación.

# Esto está en el archivo interfaz\_saludar.py  
import logica\_saludar as logica  
  
nombre = input("¿Cuál es su nombre? ")  
saludo = logica.saludar(nombre)  
print(saludo)

En la primera línea de este archivo encontramos un tipo de instrucción de Python que no habíamos usado hasta el momento:

import logica\_saludar as logica

Esta línea la usamos para *importar* un módulo para que podamos usarlo dentro de otro módulo. En este caso, estamos *importando* el módulo logica\_saludar para que podamos usar las funciones allá definidas dentro del módulo interfaz\_saludar. Adicionalmente, nuestra instrucción también está indicando que queremos hacer referencia al módulo importado usando el alias 'logica'. Como veremos a continuación esto es muy útil cuando el nombre de un módulo importado es muy largo, o cuando queremos importar módulos diferentes que tienen el mismo nombre.

Las siguientes líneas del programa son similares a las que ya habíamos estudiado excepto por la línea que dice:

saludo = logica.saludar(nombre)

La diferencia en este caso es que hemos usado el prefijo logica. para invocar la función saludar. Lo que esto significa es que se debe invocar la función saludar que está definida en el módulo que se importó con el nombre logica.

Si el módulo lo hubiéramos importado usando la instrucción

import logica\_saludar as l

la invocación a la función habría sido

saludo = l.saludar(nombre)

Finalmente, si sólo hubiéramos importado el módulo usando

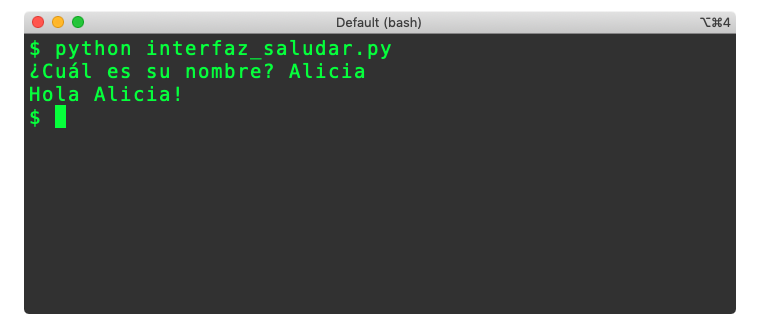
import logica\_saludar

la invocación a la función habría sido

saludo = logica\_saludar.saludar(nombre)

Esto muestra que, si no se le da un alias al módulo al importarlo, se debe utilizar el nombre completo del módulo para invocar sus funciones.

Finalmente tenemos la captura de pantalla que muestra lo que pasa cuando se ejecuta el programa 'interfaz\_saludar.py'.



## Funciones de la interfaz

En una sección anterior hablamos de la conveniencia de tener funciones sin parámetros o funciones sin retorno. En esta sección podemos retomar la discusión puesto que es precisamente en el módulo que implementa la interfaz de un programa donde tiene más sentido incluir este tipo de funciones.

### Funciones sin parámetros

Como dijimos antes, una función sin parámetros no tendría sentido dentro de la definición matemática de función. En el caso de Python, una función cuyo resultado no dependiera de ningún valor externo tampoco tendría mucho sentido; sería mucho más eficiente y claro implementarlo como una constante.

Sin embargo, dentro del módulo de la interfaz sí puede ser conveniente tener funciones sin parámetros. Tomemos como ejemplo el siguiente fragmento de código:

import libreria  
  
def calcular\_area\_cuadrado() -> float:  
 str\_lado = input("Por favor indique el lado del cuadrado: ")  
 lado = float(str\_lado)  
 area = libreria.calcular\_area(lado)  
 return area

En este programa tenemos una función sin parámetros que retorna un número decimal, pero ese número puede variar con cada ejecución. De hecho, el resultado de esta función depende completamente de algo externo: el valor que el usuario teclee cuando se ejecute la función.

Esta situación es muy frecuente dentro de los módulos que implementan la interfaz y que reciben información del usuario. Desconfíe de una función que no reciba parámetros y se encuentre mezclada con funciones que tengan que ver con la lógica de la aplicación: probablemente sea un error de diseño y sea necesario cambiar la signatura de la función.

### Funciones sin retorno

En el módulo donde se implementa la interfaz también es posible tener funciones que no tengan un retorno. Estas funciones se suelen llamar *procedimientos* porque no cumplen con la característica básica de una función de producir un resultado.

El siguiente fragmento nos muestra una función de la interfaz que no tiene un retorno:

def mostrar\_resultado(resultado: str) -> None:  
 print("El resultado es:", resultado)

Observe que a diferencia de casi todas las funciones que hemos visto hasta ahora, esta no incluye una instrucción de tipo return. Esto quiere decir que, si alguien invoca a esa función, no va a recibir un resultado. Para expresar esto en la signatura, el tipo de retorno de la función se ha declarado como None. Más adelante veremos que None se utiliza para varias cosas en Python.

El cuerpo de la función sólo tiene un llamado a la función print. Esta función, que sirve para mostrar algún texto en la consola, es la que está produciendo un resultado, aunque no sea un resultado que se pueda utilizar dentro del programa en instrucciones siguientes [[18]](#footnote-18).

## Ejercicios

1. ¿Sólo con base en la signatura y lo que sabe hasta ahora, usted pondría las siguientes funciones en el módulo de la interfaz o en la librería de su programa?

* def funcion(param1: int, param2: int) -> int
* def funcion() -> int
* def funcion(param1: int) -> None
* def funcion(param1: int) -> int

## Más allá de Python

La primera parte de esta sección introdujo algunos temas relativamente avanzados que tienen que ver con una gran área de trabajo dentro de la informática llamada *Ingeniería de Software*. La principal preocupación de esta área es cómo hacer para construir de forma eficiente software con calidad, lo cual implica considerar aspectos como la forma de estructurar el software, las metodologías de trabajo en los proyectos de desarrollo, las técnicas para diseñar interfaces que ofrezcan una buena experiencia a los usuarios, y hasta la psicología de los desarrolladores.

Es indudable que la selección de los lenguajes de programación que se usen tiene un impacto en muchos de los otros aspectos que son de interés para la Ingeniería de Software. Por ejemplo, el uso de un lenguaje como Python parece acelerar el desarrollo de programas pequeños, pero complica el mantenimiento de programas grandes debido a factores como el uso de un sistema de tipado dinámico.

Esto no quiere decir que no se pueda usar Python exitosamente para proyectos grandes. Lo que queremos decir es que la selección del lenguaje de programación debe tener en cuenta una gran cantidad de factores que van mucho más allá de cuál es el lenguaje de moda, cuál es el lenguaje que conocemos o cuál es el lenguaje que nos gustaría aprender. Todos los lenguajes tienen ventajas y desventajas que se deben sopesar cuidadosamente, y después de tomar una decisión cualquier proyecto debería introducir instrumentos para mitigar los riesgos o el impacto cuando se materialicen esos riesgos.

Por ejemplo, si se decide usar un lenguaje como Python o JavaScript para un gran proyecto, deberían definirse varias reglas desde el inicio para contrarrestar la dificultad para aplicar operaciones de *refactoring*.

En casi cualquier lenguaje de programación existen mecanismos similares a los módulos para descomponer programas y poder reutilizar bloques. Estos mecanismos también definen *espacios de nombres* (namespaces). Es decir que agrupan elementos para que los nombres no tengan que ser únicos dentro de todo el programa, y en lugar de eso sean únicos dentro de cada espacio de nombres. En Java y otros lenguajes orientados a objetos los mecanismos básicos de modularización son las clases, aunque también existen otros mecanismos como los paquetes y las librerías empaquetadas.

1. Python, al igual que otros lenguajes como Java, es realmente un lenguaje compilado e interpretado. Más aún, dependiendo de la implementación que se use, podría ser que los programas efectivamente se compilen y que no haya interpretación. Por simplicidad, en este libro hablaremos del intérprete de Python, sin entrar en detalles sobre el proceso de compilación. [↑](#footnote-ref-1)
2. Otro ejemplo es el lenguaje JavaScript, para el que diferentes compañías han construido sus propios intérpretes: SpiderMonkey para Mozilla Firefox, V8 para Google Chrome, JavaScriptCore para Safari, Chakra para Microsoft Edge y Hermes para aplicaciones Android basadas en React Native. [↑](#footnote-ref-2)
3. Por defecto, los literales de números enteros asumen que se trata de números en base 10. Sin embargo, si se preceden los números con los caracteres '0b' o '0x' significaría que se trata de números binarios o hexadecimales, respectivamente. Por ejemplo, los tres literales 0b10110, 0x16 y 22 representarían el mismo valor (el número 22 en base 10). [↑](#footnote-ref-3)
4. El texto fue tomado de <http://shakespeareshaiku.blogspot.com/> y está basado en el Soneto 141 de William Shakespeare. Versión borrador / preliminar | -------------------| Este documento es una versión preliminar para uso interno. Si encuentra algún problema o error, o si tiene algún comentario por favor repórtelo a los autores| [↑](#footnote-ref-4)
5. Si usted ha trabajado con otros lenguajes de programación, notará que en Python no se deben declarar las variables antes de usarlas, ni se debe indicar explícitamente su tipo. Esto se debe a que Python utiliza *tipado dinámico*, es decir que el intérprete infiere los tipos de variables y parámetros durante la ejecución. Veremos más adelante que, aunque en las versiones actuales no se puede *especificar* los tipos de nada, sí se pueden dar *hints* que sirven como comentarios para que los programadores puedan hacer un mejor seguimiento a los programas. [↑](#footnote-ref-5)
6. Cuando se resuelven ejercicios de matemáticas o física aparecen conceptos similares a los que estamos trabajando y es normal que se escriban expresiones muy parecidas (por ejemplo: distancia = velocidad \* x). Sin embargo, en esos contextos las expresiones no se escriben necesariamente siempre en un orden estricto, porque es un humano el que las va a interpretar. Además, porque más que asignaciones, se trata de ecuaciones que describen una equivalencia entre valores. [↑](#footnote-ref-6)
7. Aunque también está definida para números negativos, no es muy recomendable aplicar la operación módulo sobre estos números debido a que no todos los lenguajes utilizan la misma definición. Esto podría dar lugar a errores graves y difíciles de detectar. [↑](#footnote-ref-7)
8. Python, C, C++ y Java son algunos ejemplos muy populares de lenguajes de programación **imperativos**. En estos lenguajes las instrucciones de un programa sirven para darle órdenes al computador (por ejemplo, sume estos dos números, guarde el resultado en esta posición de memoria). Hay otros lenguajes de programación que siguen *paradigmas* diferentes, como el paradigma **lógico** o el **funcional**. En esos lenguajes las instrucciones de un programa no son órdenes sino expresiones lógicas que el computador debe resolver, o definiciones de funciones que el computador debe evaluar, por ejemplo. Después de adquirir una cierta destreza con algún lenguaje de programación particular es muy recomendable estudiar algún lenguaje basado en un paradigma diferente: está demostrado que aprender a resolver los mismos problemas utilizando herramientas y métodos completamente diferentes es parte de lo que hace a un experto. [↑](#footnote-ref-8)
9. Usamos acá el término *parámetro* para evitar confusiones que podrían aparecer si usáramos el término *variable*. [↑](#footnote-ref-9)
10. En realidad, int, float y str son más que simples funciones. Sin embargo, para poder explicar todos sus detalles tendríamos que discutir antes lo que significa programación orientada a objetos y los conceptos de clase y constructor (que no se estudiarán en este libro). Para efectos prácticos, podemos suponer que se trata de 3 funciones y podemos utilizarlas como tal sin ningún problema. [↑](#footnote-ref-10)
11. La función int también puede invocarse con un parámetro nombrado llamado base que tiene por defecto el valor 10. Este parámetro adicional sirve para indicar la base en la que está expresado el número, de tal forma que podamos hacer con facilidad conversiones de números. Más adelante estudiaremos los conceptos de parámetros nombrados y de parámetros con valor por defecto. [↑](#footnote-ref-11)
12. Más adelante estudiaremos usos más avanzados de la función que permiten, por ejemplo, separar los elementos usando un caracter diferente. [↑](#footnote-ref-12)
13. Otro caracter interesante es el que tiene el número 7 y que es llamado "beep": cuando se *imprime* este caracter en la consola, el computador debería emitir un anuncio sonoro. En sistemas operativos modernos, el sonido utilizado es el sonido de alerta configurado. [↑](#footnote-ref-13)
14. Una palabra reservada significa que es una palabra que nosotros no podemos usar en nuestros programas para nuestros identificadores. Por ejemplo, no podemos tener ni una función ni una variable que se llamen def. Python tiene varias palabras reservadas que iremos descubriendo, como return, del, import, pass, y raise, entre otras. [↑](#footnote-ref-14)
15. La regla que acabamos de ilustrar (separar palabras usando el símbolo \_) no es una regla sintáctica del lenguaje, sino es una regla de *estilo*. Es decir, nosotros podríamos haber llamado a la función areaCuadrado y el programa habría funcionado igual, pero no habría respetado las reglas de estilo que sirven para hacer los programas más legibles y consistentes. Si usted ha utilizado algún otro lenguaje de programación, es posible que el estilo al que esté acostumbrado le sugiera usar nombres como areaCuadrado o AreaCuadrado. [↑](#footnote-ref-15)
16. Las preocupaciones de las que hemos estado hablando se conocen usualmente como *Requerimientos No Funcionales* o como *Atributos de Calidad*. Aunque usualmente son un tema que se estudia más adelante, no está de más empezar a usar esta terminología y empezar a tener una sensibilidad hacia estas problemáticas. Pensar únicamente en la corrección del software y en construir programas lo más rápidamente posible nos ha llevado a una situación en la que tenemos cada vez más software que es inútil y tiene que botarse a la caneca y reconstruirse. [↑](#footnote-ref-16)
17. Si se compara el tiempo que toma el desarrollo inicial de un software y el tiempo que dura su mantenimiento posterior, en la mayoría de casos de software exitoso se va a encontrar que el mantenimiento se lleva la mayor parte del tiempo. Por ejemplo, el software que utilizan muchos bancos del mundo en su *core*, o el software que usan las aerolíneas para el control de vuelos y reservas, o el software que utilizan las empresas de telecomunicaciones para hacer conexiones entre redes, se construyó hace decenas de años, usando tecnologías que hoy ni siquiera se enseñan ya. La tecnología y las necesidades, especialmente de integración, han cambiado, lo cual ha obligado a hacerles actualizaciones permanentes a estos sistemas: si no fuera porque estos sistemas se construyeron para garantizar su mantenibilidad, hace muchos años que habrían tenido que ser remplazados y las empresas que los usan habrían tenido que incurrir en costos altísimos que podrían hasta haberlas quebrado. [↑](#footnote-ref-17)
18. Esto es lo que se conoce como un efecto de borde de un programa. Otros efectos de borde típicos son escribir en un archivo y enviar un mensaje por la red. [↑](#footnote-ref-18)