 On this page

Nivel 0

Antes de empezar



1. Introducción

The computer programmer is a creator of universes for which he alone is the lawgiver.

—Joseph Weizenbaum

By Mario Sánchez
© Copyright Agosto de 2020.

[Departamento de Ingeniería de Sistemas y Computación](#) - Universidad de los Andes

1.1. ¿Qué es programar?

[Print to PDF](#)

Programar es la acción de escribir un conjunto instrucciones (un programa) para que un computador las ejecute. Por ejemplo, el siguiente es el primer programa que la mayoría de programadores escribe:

```
print('Hola, mundo!')
```

Cuando alguien decide ejecutar (o correr) este programa, el computador va a mostrar en la pantalla un mensaje que dice [Hola, mundo!](#).

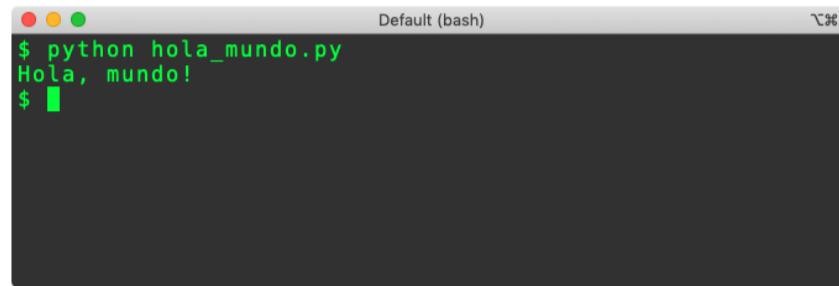


Fig. 1.1 Ejecución del primer programa

Aunque sencillo, este programa nos permite ilustrar varias ideas importantes. En primer lugar, el programa se escribió usando el lenguaje de programación Python. Así como en español tenemos reglas de gramática y un conjunto de palabras para combinar (vocabulario), un lenguaje de programación tiene reglas de sintaxis, una colección de librerías con programas que se pueden reutilizar (el vocabulario) y unas herramientas que permiten ejecutar los programas que se escriban con ese lenguaje. En nuestro ejemplo, estamos siguiendo las reglas de sintaxis de Python, usamos una función llamada [print](#) y ejecutaremos el programa usando el intérprete [1] de Python [2].

Programar no es en sí misma una actividad difícil, pero sí es una actividad compleja que involucra varios elementos y habilidades complementarias. La más importante posiblemente sea la habilidad para resolver problemas, que en realidad es una combinación de otras habilidades como comprensión de lectura, abstracción, descomposición, razonamiento abstracto y creatividad. Con esto queremos decir que, para programar, es necesario ser capaz de resolver problemas independientemente de que haya tecnología y computadores involucrados.

Otro componente para resaltar es el pensamiento algorítmico, el cual hace referencia a una cierta forma de aproximarse a la resolución de problemas. El término algoritmo hace referencia a las instrucciones para resolver un determinado problema. Un ejemplo común de algoritmo son las instrucciones para fritar un huevo, o las instrucciones que se encuentran en el envase de un Shampoo. Pensamiento algorítmico hace referencia a la habilidad para describir cómo se debe llegar a la solución de un problema en términos de una serie de instrucciones que alguien más pueda repetir.

En nuestras vidas resolvemos problemas todo el tiempo (manejar un carro, entrar a nuestra oficina, pagar los recibos de servicios públicos, organizar nuestra casa), pero no necesariamente somos capaces de explicarle a alguien más cada uno de estos procesos con el nivel de detalle suficiente para que los pueda hacer exactamente igual. Cuando empecemos a programar nos daremos cuenta que uno de los obstáculos más grandes va a ser vencer la tentación de darle al computador órdenes demasiado complejas. Por el contrario, programar requiere usar órdenes relativamente sencillas que permitan ir resolviendo el problema (en lugar de decirle a alguien 'frita un huevo', le vamos a tener que decir 'prende el fogón de la estufa, pon una cacerola con mantequilla, espera a que se derrita la mantequilla, ...')

Programar requiere también conocer al menos un lenguaje de programación para poder escribir las instrucciones para el computador. Posiblemente aprender un lenguaje de programación sea la parte más sencilla de todo el proceso, pero es la que fácilmente se confunde con la parte central de aprender a programar. Esto es análogo a pensar que para pintar al óleo lo más importante es aprender los nombres de los colores y conocer la técnica para usar los pinceles: seguramente alguien que no tenga esos conocimientos básicos encontrará dificultades para pintar, pero con seguridad otras habilidades y actitudes son más importantes para producir verdaderas obras de arte.

Este libro pretende hacer explícito que *aprender a programar no es lo mismo que aprender un lenguaje de programación*. Tuvimos que escoger un lenguaje para nuestros ejemplos (Python), pero casi todos los conceptos que veremos podrían aplicarse a otros lenguajes de programación. En lo posible, haremos explícito cuando algo de lo que expliquemos sea exclusivo de Python.

Finalmente, programar requiere una buena disposición hacia la tecnología y hacia el auto-aprendizaje. En este libro usaremos un conjunto de herramientas y librerías relativamente sencillas que además será explicado en detalle. Estas deberían ser suficientes para construir desde programas triviales hasta programas muy interesantes y útiles. Pero un buen programador debería ser capaz (¡y debería tener muchas ganas!) de aprender a usar nuevas tecnologías por su propia cuenta.

-
- [1] Más adelante vamos a explicar qué significa el intérprete. Por ahora es suficiente con saber que un intérprete es un programa capaz de ejecutar otros programas. En este caso, el intérprete será capaz de ejecutar programas escritos usando Python.
 - [2] Si en lugar de Python hubiéramos usado el lenguaje de programación C, el programa que habríamos tenido que escribir sería el siguiente: `main() { printf("hello, world\n"); }`. Además, el programa no lo ejecutaríamos usando el intérprete de Python.
-

By Mario Sánchez
© Copyright Agosto de 2020.

[Departamento de Ingeniería de Sistemas y Computación](#) - Universidad de los Andes

1.2. Prácticas fundamentales

[Print to PDF ▶](#)

La experiencia de varios años enseñando a programar nos muestra que con una adecuada *actitud* y compromiso cualquiera puede aprender a programar. Los estudiantes que se aproximan a la programación esperando cumplir con un requisito, sin ver el enorme potencial que podría tener para su desempeño profesional, usualmente tienen menos éxito que aquellos que tienen una mente abierta y curiosa y que se enfrenten a cada reto esperando aprender algo de él.

Además de esto, la experiencia también nos ha mostrado que los estudiantes que realizan las siguientes 3 acciones usualmente tienen mucho más éxito en su proceso de aprendizaje, disfrutan más la experiencia y siguen aprendiendo a programar más allá de su primer curso.

1.2.1. Lectura con atención

Para poder programar es necesario leer con atención todo lo relacionado con el problema que se esté solucionando. Es casi imposible construir una buena solución si no se han leído con cuidado las condiciones de lo que se está pidiendo ni las restricciones para la solución.

Muchas veces los problemas a los que se enfrentan los estudiantes que empiezan a programar no tienen que ver con la programación en sí misma, sino que tienen que ver con que el estudiante no entendió el problema que tenía que resolver.

¡Nunca empiece a programar sin entender antes lo que le están preguntando!

1.2.2. Práctica deliberada y reflexiva

The only way to learn a new programming language is by writing programs in it.

—Dennis Ritchie, creador de C y Unix

Al igual que cualquier otra actividad basada en habilidades, programar requiere practicar. Así como no se puede aprender a tocar violín o a montar bicicleta leyendo todos los libros disponibles sobre el tema, para aprender a programar se necesita practicar programando.

Más aún, para hacer más eficiente su proceso de aprendizaje, un estudiante de programación debería esforzarse por hacer una práctica deliberada y reflexiva. Práctica deliberada hace referencia a tener un objetivo específico cuando se practica. Por ejemplo, cuando un futbolista practica no se limita a *jugar fútbol*, sino que en cada sesión repite ejercicios diseñados para ayudarlo a desarrollar una determinada habilidad. De igual forma, en cada sesión de práctica los pianistas más exitosos definen un objetivo particular (practicar un tipo de técnica, resolver un fragmento de una pieza) en lugar de simplemente sentarse a *tocar piano*.

Por otro lado, práctica reflexiva hace referencia al proceso que debería hacer un estudiante al terminar una práctica. En lugar de simplemente dar el ejercicio por terminado, el estudiante debería tomarse un momento para reflexionar sobre lo que hizo, lo que aprendió, los problemas que enfrentó y las conclusiones que se podrían sacar de la experiencia. Se ha visto en diversas situaciones que el esfuerzo invertido en este proceso de reflexión hace que la práctica sea mucho más efectiva y termina reduciendo el esfuerzo total que se debe hacer.

Este libro incluye numerosos ejercicios, seleccionados para ejercitarse habilidades particulares relacionadas con cada uno de los temas. A medida que vaya avanzando, procure resolver los ejercicios haciendo una reflexión sobre lo que aprendió al final de ellos.

1.2.3. Lectura de código

Al igual que un pintor no podría pintar sus propias obras sin haber visto las de otros, o un escritor no podría escribir una novela sin haber leído las de muchos otros, para escribir programas es necesario poder leer programas escritos por otros. Sin embargo, no se trata de hacer una lectura superficial, sino de hacer una lectura cuidadosa que nos permita identificar las características y objetivos de cada uno y nos permita confrontar nuestras propias dudas y vacíos en nuestro conocimiento.

A lo largo de este libro encontrará numerosos fragmentos de código que ilustran conceptos particulares. Además, encontrará también programas más largos y complejos que retarán sus habilidades de lectura. ¡Haga el esfuerzo de leer estos programas, así no los entienda completamente en un primer momento! En el nivel 1 le daremos algunas

 On this page

[1.2.1. Lectura con atención](#)

[1.2.2. Práctica deliberada y reflexiva](#)

[1.2.3. Lectura de código](#)

recomendaciones adicionales sobre cómo leer código.

By Mario Sánchez
© Copyright Agosto de 2020.

[Departamento de Ingeniería de Sistemas y Computación](#) - Universidad de los Andes

 On this page[Print to PDF](#) ▶

1.3. Sobre este libro

Este libro está pensado para acompañar el curso de Introducción a la Programación del [Departamento de Ingeniería de Sistemas](#) de la Universidad de los Andes, el cual está dirigido a estudiantes que nunca hayan programado. Pensando en esto, el libro tiene las siguientes características.

- Es un libro básico para aprender a programar: A diferencia de otros libros que le enseñan Python a programadores que ya conozcan otros lenguajes, este libro no supone ningún conocimiento previo de programación. Si usted ya sabe programar en algún otro lenguaje, es posible que este libro sea *demasiado* básico para usted.
- Libro soporte: El libro está pensado para *soportar* y aclarar cualquier duda que haya quedado en el aula. Un estudiante del curso debería poder encontrar acá aclaraciones y ejercicios adicionales sobre cualquier tema. Sin embargo, el libro no siempre seguirá el mismo orden que se lleve en el aula de clase: en algunas secciones el libro presentará temas de forma completa desde el inicio, entrando en un mayor nivel de detalle del que normalmente se estudiaría la primera vez que se encontrara ese tema.
- Python: El curso está basado en Python, así que el libro también está basado en Python. Sin embargo, el libro hace un esfuerzo por identificar claramente los conceptos centrales que son aplicables a otros lenguajes de programación en lugar de presentarlos como características del lenguaje.
- Organización: El libro está organizado siguiendo la estructura del curso y el curso está organizado teniendo en cuenta restricciones y necesidades que son propias de nuestros programas académicos: la cantidad de tiempo disponible, el contenido de los cursos siguientes, los programas en los que están inscritos nuestros estudiantes, y la necesidad de desarrollar competencias algorítmicas por encima de competencias para la estructuración de programas. Esto hace que la organización de este libro sea muy diferente a la de muchos otros libros de programación con Python. Por ejemplo, el tema de programación orientada a objetos, que aparece en los primeros capítulos de otros libros, está completamente ausente en este. Algo similar pasa con el tema de manejo y creación de excepciones.

By Mario Sánchez
© Copyright Agosto de 2020.

[Departamento de Ingeniería de Sistemas y Computación](#) - Universidad de los Andes

1.4. El lenguaje de programación Python

[Print to PDF](#)

Simple is better than complex.

—Tim Peters, The Zen of Python

≡ On this page

[1.4.1. Python](#)

[1.4.2. Python para Introducción a la Programación](#)

Como ya dijimos, para este libro (y para el curso de Introducción a la Programación del [Departamento de Ingeniería de Sistemas](#) de la Universidad de los Andes), el lenguaje de programación que usaremos es Python o más precisamente Python 3 [1].

Python es uno de miles de lenguajes de programación existentes en el mundo. Es posible que usted haya escuchado hablar o haya usado alguno de los siguientes: C, C++, Java, Kotlin, Scala, JavaScript, TypeScript, C#, PHP, Ruby, Visual Basic, Pascal, Basic, Logo, Cobol, Fortran, LISP. Además, hay ambientes de trabajo especializados que tienen sus propios lenguajes de programación, como R (para estadística), MathLab (para aplicaciones científicas) o Swift (para aplicaciones en iOS y MacOS).

Seleccionar un lenguaje de programación no es una tarea fácil: todos tienen ventajas y desventajas que van mucho más allá de lo que puede y no puede hacerse. Algunos lenguajes están diseñados o son particularmente buenos para construir cierto tipo de aplicaciones (por ejemplo, JavaScript para aplicaciones que corren en navegadores web, Swift para aplicaciones en iOS, Visual Basic para aplicaciones en Office, C/C++ para aplicaciones de bajo nivel o que requieran un altísimo desempeño) así que sería difícil usarlos en otros contextos o reemplazarlos por otros. Más allá de esto, casi cualquier lenguaje puede utilizarse para resolver cualquier problema.

Otra diferencia importante entre lenguajes es la disponibilidad de ayuda (manuales, ejemplos, foros, cursos, etc.), librerías (programas que resuelvan problemas específicos que podamos reutilizar en nuestros propios programas) y herramientas de desarrollo. Seguramente será más difícil empezar a usar un lenguaje *esotérico*, usado por pocas personas en el mundo, que un lenguaje muy popular para el cual haya una gran cantidad de recursos disponibles y un público creciente que produzca cada vez más material de soporte. Sin embargo, debe ser claro que popularidad no implica calidad: hay lenguajes que han sido extraordinariamente populares a pesar de la gran cantidad de problemas de fondo que tenían. Además, la popularidad cambia con el tiempo: los lenguajes que hoy en día encabezan las listas eran prácticamente desconocidos hace 10 años y posiblemente serán remplazados dentro de los siguientes 10 años.

Finalmente, hay diferencias importantes en la complejidad misma de los lenguajes y de los programas que se construyen con ellos. El programa que en un lenguaje puede requerir unas pocas líneas de código, en otro lenguaje puede requerir diez o más veces la cantidad de líneas de código [2]. Además, en algunos lenguajes pueden escribirse programas crípticos, cuyo funcionamiento puede ser imposible de entender sin ayuda del autor, mientras que otros lenguajes fomentan que los programas sean sencillos y fáciles de entender y de mantener, sin sacrificar las funcionalidades.

1.4.1. Python

Comparado con otros lenguajes, Python tiene algunas limitaciones evidentes, pero que no son relevantes para este libro. Por el contrario, Python tiene varias características que lo hacen un lenguaje muy adecuado, en este momento, para aprender a programar. La más importante es que Python comparte muchos conceptos y estructuras con muchos otros lenguajes, así que debería ser fácil aprender uno de estos lenguajes después de aprender a programar en Python.

La segunda razón es que la filosofía detrás del diseño del lenguaje buscaba la simplicidad y la claridad. El texto “The Zen of Python”, de Tim Peters, resume esta filosofía en 19 aforismos de los cuales quisiéramos destacar los siguientes tres porque deberíamos aplicarlos a nuestros propios programas:

- Explicit is better than implicit (Explícito es mejor que implícito). ¿Recuerda cuando su profesor de matemáticas en el colegio le decía que el procedimiento para resolver un examen tenía que quedar escrito? Con esto su profesor buscaba entender qué era lo que estaba haciendo y cómo estaba llegando a la respuesta. Si en algún punto del procedimiento usted tenía un error, su profesor probablemente le corrigió el error y siguió aplicando el procedimiento para llegar a la respuesta correcta. Lo mismo ocurre al programar: es mejor hacer explícito lo que se está haciendo, en pequeños pasos, para que sea más fácil encontrar errores o implementar nuevas funcionalidades.
- Simple is better than complex (Simple es mejor que complejo). Cuando tenga que resolver un problema, le recomendamos que busque la solución más simple que pueda funcionar: entre más complicada sea una solución, más difícil será construirla, corregirla o mejorarlala.

- Readability counts (La facilidad de lectura es importante). Parafraseando al profesor Harold Abelson, los programas se deben construir pensando primero en que humanos van a leerlos y luego en que un computador va a ejecutarlos. Además, el primer humano que va a leer nuestros programas somos nosotros mismos: deberíamos hacernos el favor de siempre escribir código que podamos leer con facilidad.

💡 Tip

En este libro vamos a incluir frecuentemente anotaciones para reforzar estas ideas y, dentro del nivel 1, una sección dedicada enteramente a estos temas. Por favor no los olvide.

Para acceder al texto completo de “The Zen of Python” basta escribir `import this` en un intérprete de Python. El resultado debería ser el siguiente:

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

1.4.2. Python para Introducción a la Programación

Para concluir esta sección, vamos a resumir los principales motivos detrás de la decisión que tomamos de usar el lenguaje Python en el curso Introducción a la Programación de la Universidad de los Andes.

1. Facilidad. El primer punto tuvo que ver con las características del lenguaje que lo hacen fácil de aprender para los que nunca antes han programado. En particular, para nosotros era importante que no fuera obligatorio utilizar los conceptos de programación orientada a objetos para poder construir programas interesantes y relativamente complejos.
2. Uso actual en cursos posteriores. Introducción a la Programación es el primer curso donde los estudiantes de nuestra Universidad tienen que programar, pero no es el único. Cada vez más programas y facultades tienen cursos donde la programación es un elemento central y en varios de estos cursos Python ya había sido adoptado.
3. Compatibilidad. Como dijimos antes, Python se basa en conceptos que son comunes a muchos lenguajes de programación, así que la transición a otros lenguajes debería ser relativamente fácil. Nosotros creemos que al utilizar Python vamos a lograr que nuestros estudiantes desarrollen las habilidades de programación fundamentales que les permitan desempeñarse satisfactoriamente en tanto en cursos posteriores como en sus profesiones.

[1] Desafortunadamente, Python tiene una historia larga de problemas de versiones. En Diciembre de 2008 apareció oficialmente Python 3.0, lo cual debería haber logrado que las versiones anteriores fueran consideradas *obsoletas*. El problema es que había tantos programas y librerías construidas sobre las versiones 2.x que la transición completa ha tardado más de 10 años en ocurrir. El resultado de esto es que todavía sea fácil encontrar instalaciones de Python 2.7 como por ejemplo en las instalaciones por defecto del sistema operativo MacOS. Para efectos de este libro, cuando corra el intérprete de Python, tenga siempre cuidado de que sea una versión mayor a la 3.6.

[2] Un buen ejemplo de esto se encuentra en [The Hello World Collection](#), donde hay más de 600 implementaciones del programa para escribir “Hola, Mundo!”

 On this page

1.5. Estructura del libro

[Print to PDF](#) ▶

El libro está organizado en 4 grandes partes que corresponden a los 4 niveles en los que está organizado nuestro curso Introducción a la Programación. Los 4 niveles y sus temas principales son los siguientes:

1. Descubriendo el mundo de la programación. En esta parte se introducen los conceptos básicos de la programación y se empieza a trabajar el tema del uso de funciones y módulos para descomponer y estructurar adecuadamente los programas.
2. Tomando decisiones. En esta parte se introducen las instrucciones condicionales y una estructura de datos que nos permite manejar información un poco más compleja (diccionarios).
3. Repetir acciones. A partir de esta sección los programas que se pueden construir se vuelven mucho más poderosos porque se estudian las instrucciones repetitivas y se introducen las estructuras de datos lineales.
4. Solucionar problemas con matrices y librerías. La última sección del libro presenta una estructura de datos nueva (matrices) y luego se concentra en el uso de una librería muy utilizada para el procesamiento de datos (*Pandas*).

Los 4 niveles son incrementales: lo que se aprende en un nivel se utiliza y se refuerza en el siguiente, de tal forma que al final todos los temas se hayan estudiado y practicado varias veces.

Además, el libro incluye también secciones en las que se tratan temas metodológicos que son transversales a los 4 niveles, y secciones con referencias técnicas de algunos elementos del lenguaje Python.

By Mario Sánchez
© Copyright Agosto de 2020.

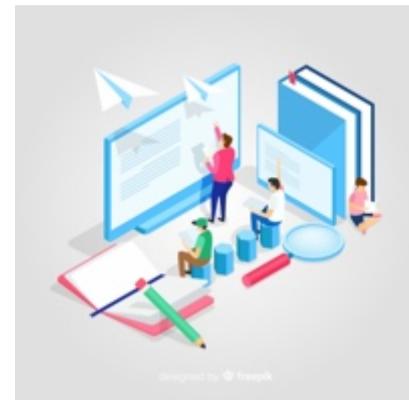
[Departamento de Ingeniería de Sistemas y Computación](#) - Universidad de los Andes

 On this page

1.6. Créditos de las imágenes

Este libro contiene algunas imágenes que han sido creadas por terceros y que han sido puestas a disposición para su uso, condicionado a que se les reconozca su autoría. Las imágenes que no se encuentran en esta sección son imágenes de nuestra propia elaboración.

Las siguientes son las imágenes de terceros usadas con autorización en este libro y sus autores.



Designed by [Freepik](#)



Foto tomada por [Darren Coleshill](#) en [Unsplash](#)

By Mario Sánchez
© Copyright Agosto de 2020.

[Departamento de Ingeniería de Sistemas y Computación](#) - Universidad de los Andes

2.1. Un programa para leer

[Print to PDF ▶](#)[On this page](#)[2.1.1. Por qué leer código de alguien más](#)[2.1.2. El primer programa en Python](#)

⚠ 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.

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.

2.1.1. 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.

It's harder to read code than to write it.

—Joel Spolsky

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 un conjunto de estrategias que usará para resolver los problemas que encuentre más adelante. 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.

2.1.2. 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.

1. 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.
2. Note que hay algunas palabras que se repiten.
3. Note también que hay algunas cosas escritas en español y otras escritas en inglés.
4. Intenta 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.
5. Anote qué cosas del programa no sabe qué significan.

```
1 def perimetro_triangulo(cateto1: float, cateto2: float)->float:
2     hipotenusa = calcular_hip(cateto1, cateto2)
3     return cateto1 + cateto2 + hipotenusa
4
5
6 def calcular_hip(cateto1: float, cateto2: float)->float:
7     suma_cuadrados = (cateto1 ** 2) + (cateto2 ** 2)
8     hipotenusa = pow(suma_cuadrados, 0.5)
9     return hipotenusa
10
11
12 cadena_cat_1 = input("Indique la longitud del primer cateto: ")
13 cadena_cat_2 = input("Indique la longitud del segundo cateto: ")
14 cat_1 = float(cadena_cat_1)
15 cat_2 = float(cadena_cat_2)
16
17 perimetro = perimetro_triangulo(cat_1, cat_2)
18
19 print("El perímetro de un triángulo rectángulo que tenga catetos de longitud",
20       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.

```

1 # Este programa está escrito en el archivo perimetro.py
2
3 def perimetro_triangulo(cateto1: float, cateto2: float)->float:
4     """
5         Esta función calcula el perímetro de un triángulo rectángulo
6         dada la longitud de sus dos catetos
7         Parámetros:
8             cateto1 (float): la Longitud del primer cateto del triángulo
9             cateto2 (float): la Longitud del segundo cateto del triángulo
10            Retorno
11                (float): la longitud del perímetro del triángulo
12        """
13
14    # Usar la función calcular_hip para calcular la longitud del lado faltante
15    hipotenusa = calcular_hip(cateto1, cateto2)
16
17    # Sumar los tres lados y convertirlos en la respuesta de la función
18    return cateto1 + cateto2 + hipotenusa
19
20
21 def calcular_hip(cateto1: float, cateto2: float)->float:
22     """
23         Esta función calcula la longitud de la hipotenusa en un triángulo rectángulo
24         dada la longitud de sus dos catetos
25         Parámetros:
26             cateto1 (float): la Longitud del primer cateto del triángulo
27             cateto2 (float): la Longitud del segundo cateto del triángulo
28             Retorno
29                 (float): la longitud de la hipotenusa
30        """
31
32    # Sumar la longitud de los catetos elevados al cuadrado
33    suma_cuadrados = (cateto1 ** 2) + (cateto2 ** 2)
34
35    # Calcular la raíz cuadrada de la suma usando la función pow y el exponente 0.5
36    hipotenusa = pow(suma_cuadrados, 0.5)
37
38    # Solicitarle al usuario la longitud de los dos catetos
39    cadena_cat_1 = input("Indique la longitud del primer cateto: ")
40    cadena_cat_2 = input("Indique la longitud del segundo cateto: ")
41
42    # Convertir los caracteres dados por el usuario en un número decimal
43    cat_1 = float(cadena_cat_1)
44    cat_2 = float(cadena_cat_2)
45
46    # Llamar a la función con los valores recibidos
47    perimetro = perimetro_triangulo(3,4)
48
49    # Mostrarle al usuario el resultado de la operación
50    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?
- ¿Cuál es la diferencia entre lo que está escrito en español y lo que está escrito en inglés?

By Mario Sánchez
 © Copyright Agosto de 2020.

[Departamento de Ingeniería de Sistemas y Computación](#) - Universidad de los Andes

2.2. Ambiente básico de trabajo

[Print to PDF](#)

On this page

- [2.2.1. IDE - Integrated Development Environment](#)
- [2.2.2. Intérprete](#)
- [2.2.3. REPL para Python](#)
- [2.2.4. Línea de comandos / terminal / consola](#)
- [2.2.5. Otras herramientas](#)
- [2.2.5.1. Manejador de paquetes](#)
- [2.2.5.2. Notebooks](#)

⚠ 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.

ℹ Objetivo de la sección

El objetivo de esta sección es presentar las principales herramientas que se necesitan para poder programar, para que usted pueda reconocer sus nombres y buscar las que le hagan falta.

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.

2.2.1. IDE - Integrated Development Environment

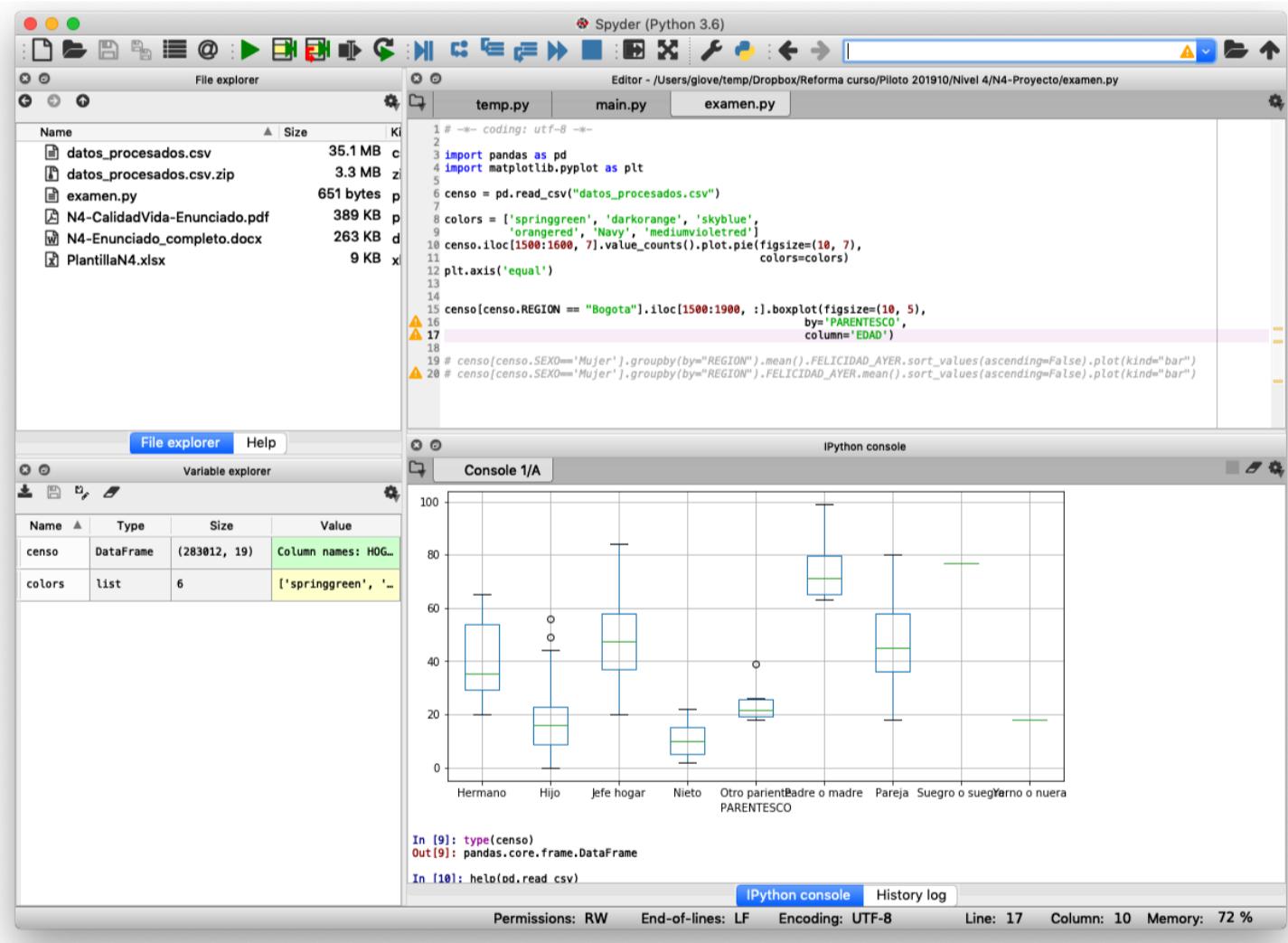


Fig. 2.1 Captura de pantalla de Spyder

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.
- Tiene un intérprete de Python bien integrado.
- 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.

Otros IDE populares que están disponibles en este momento para desarrollar programas en Python incluyen Visual Studio Code, PyCharm, Eclipse (+PyDev) y VIM. Si tiene la oportunidad de escoger el IDE que va a usar, asegúrese de entender las capacidades que tenga (por ejemplo para completar código y hacer *debugging*), la compatibilidad con otras herramientas, y el tipo de licencia que esté disponible.

2.2.2. Intérprete

```
Default (bash)
$ python hola_mundo.py
Hola, mundo!
$
```

Fig. 2.2 Ejecución del primer programa

Python es un lenguaje interpretado [1]: esto significa que para correr los programas escritos en Python es necesario que otro programa llamado intérprete los ejecute. En la [figura 2.2](#) se puede ver que se corrió un programa que se escribió en el archivo `hola_mundo.py` usando el intérprete de Python que se invocó con el comando `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].

2.2.3. REPL para Python

```
Default (bash)
osx$ python
Python 3.6.3 (default, Oct  4 2017, 06:09:15)
[GCC 4.2.1 Compatible Apple LLVM 9.0.0 (clang-900.0.37)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello, World!")
Hello, World!
>>> nombre = input("¿Cuál es su nombre? ")
¿Cuál es su nombre? Juan
>>> print("Bienvenido " + nombre)
Bienvenido Juan
>>> 2 + 1.14159
3.14159
>>> 145 * 5
725
>>> 145 ** 5
64097340625
>>> exit()
osx$
```

Fig. 2.3 REPL de Python

Como en el caso de otros lenguajes interpretados, Python 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 estándar 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 estándar 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.

El otro REPL para Python ampliamente utilizado se llama IPython y es el que está embebido dentro de Spyder.

También puede ejecutarse desde la línea de comandos usando el comando `ipython`. Aunque IPython tiene algunas ventajas sobre el REPL normal, no son realmente significativas cuando apenas se está aprendiendo a programar.

Usted reconocerá que estamos usando IPython en lugar del REPL normal porque en lugar de los carácter `>>>` aparece el número de la instrucción que se está ejecutando y se separan las instrucciones ingresadas (`IN`) del resultado de la ejecución (`OUT`):

```
osx$ ipython
Python 3.7.7 (default, Mar 10 2020, 15:43:33)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.15.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: valor = 1 + 1

In [2]: print(type(valor))
<class 'int'>
In [3]: valor
Out[3]: 2
In [4]:
```

Fig. 2.4 Captura de pantalla de IPython

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.

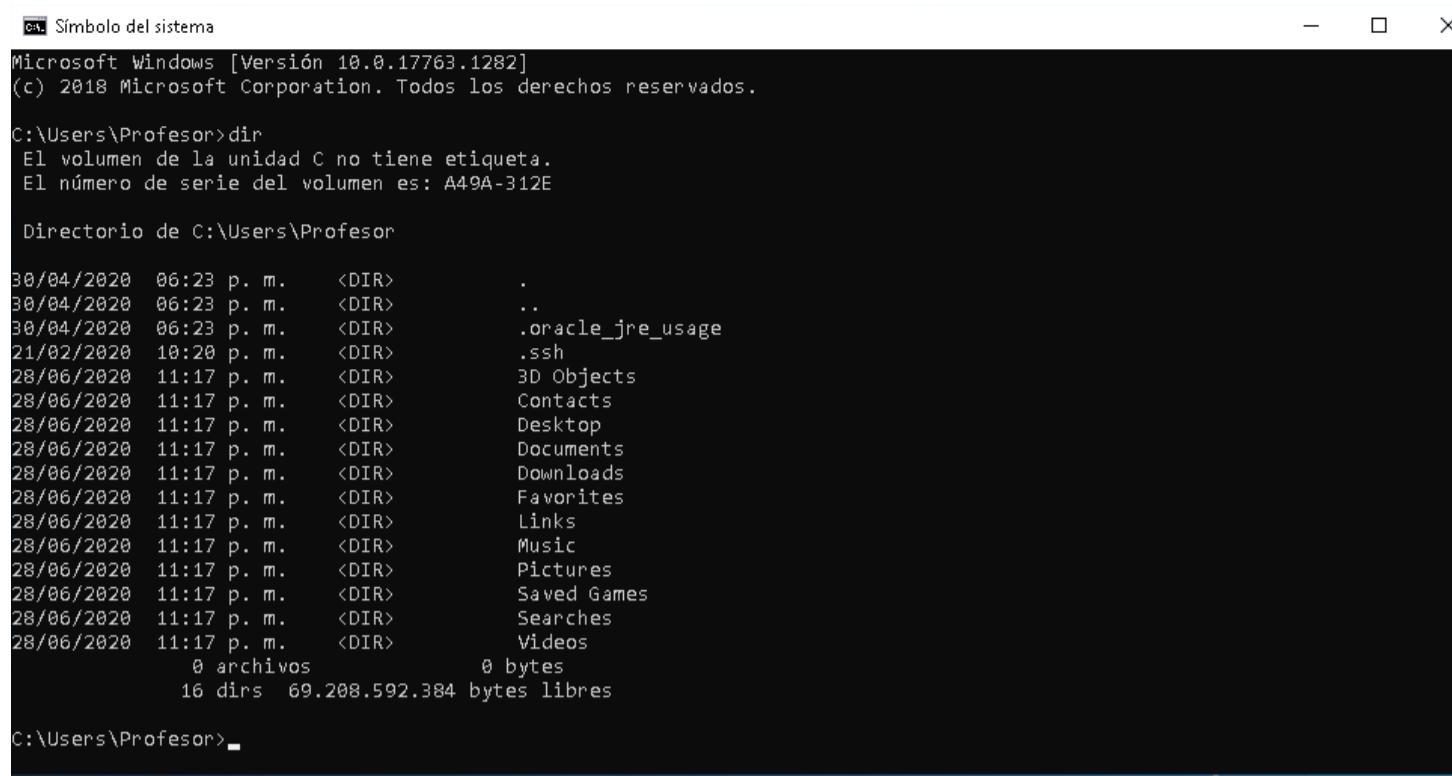
2.2.4. Línea de comandos / terminal / consola

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 las líneas de comandos de cada sistema operativo, sino mostrarle que existen e invitarlo para que estudie por su cuenta su funcionamiento.

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



```

Símbolo del sistema
Microsoft Windows [Versión 10.0.17763.1282]
(c) 2018 Microsoft Corporation. Todos los derechos reservados.

C:\Users\Profesor>dir
El volumen de la unidad C no tiene etiqueta.
El número de serie del volumen es: A49A-312E

Directorio de C:\Users\Profesor

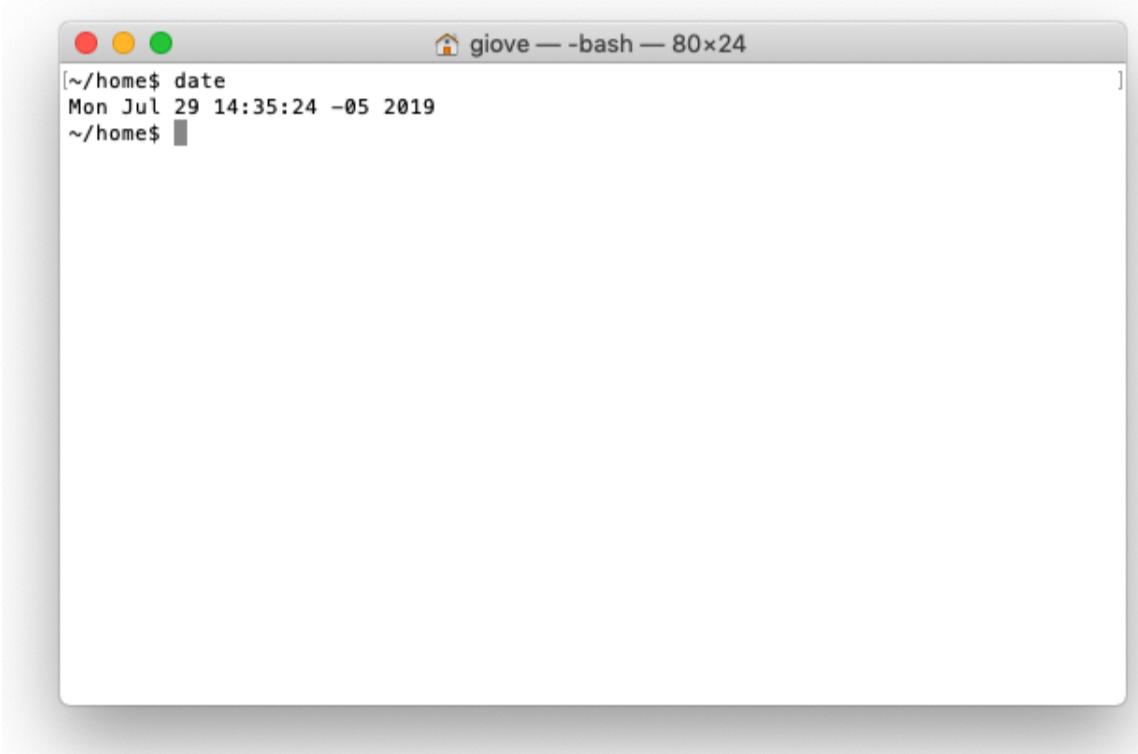
30/04/2020 06:23 p. m. <DIR> .
30/04/2020 06:23 p. m. <DIR> ..
30/04/2020 06:23 p. m. <DIR> .oracle_jre_usage
21/02/2020 10:20 p. m. <DIR> .ssh
28/06/2020 11:17 p. m. <DIR> 3D Objects
28/06/2020 11:17 p. m. <DIR> Contacts
28/06/2020 11:17 p. m. <DIR> Desktop
28/06/2020 11:17 p. m. <DIR> Documents
28/06/2020 11:17 p. m. <DIR> Downloads
28/06/2020 11:17 p. m. <DIR> Favorites
28/06/2020 11:17 p. m. <DIR> Links
28/06/2020 11:17 p. m. <DIR> Music
28/06/2020 11:17 p. m. <DIR> Pictures
28/06/2020 11:17 p. m. <DIR> Saved Games
28/06/2020 11:17 p. m. <DIR> Searches
28/06/2020 11:17 p. m. <DIR> Videos
28/06/2020 11:17 p. m. <DIR> .
          0 archivos           0 bytes
          16 dirs  69.208.592.384 bytes libres

C:\Users\Profesor>_

```

Fig. 2.5 Línea de comandos en Windows: cmd

En MacOS X, la línea de comandos se invoca corriendo el programa [Terminal](#).

**Fig. 2.6** Línea de comandos en MacOS X: Terminal

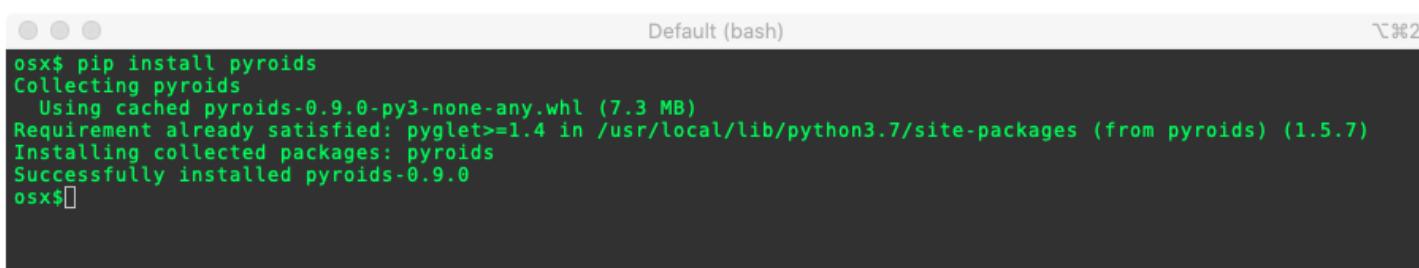
En Linux, la línea de comandos suele estar corriendo todo el tiempo, pero dependiendo de la distribución que esté usando se llega a ella de formas diferentes.

2.2.5. Otras herramientas

2.2.5.1. Manejador de paquetes

En el mundo de la programación, una de las cosas más complejas de manejar son las dependencias hacia otros programas. En este libro veremos cómo nuestros programas poco a poco van a volverse más complicados y vamos a empezar a integrar programas construidos por otros desarrolladores. Para lidiar con esa complejidad existen herramientas llamadas *manejadores de paquetes*.

En el mundo de Python hay dos manejadores de paquetes que se usan principalmente. El primero se llama [pip](#) y es capaz de buscar, instalar, actualizar y desinstalar paquetes disponibles en el Python Package Index <https://pypi.org/>. Como se ve en la [figura 2.7](#), usando [pip](#) es posible instalar un paquete y todas sus dependencias utilizando un solo comando.



```
osx$ pip install pyroids
Collecting pyroids
  Using cached pyroids-0.9.0-py3-none-any.whl (7.3 MB)
Requirement already satisfied: pyglet>=1.4 in /usr/local/lib/python3.7/site-packages (from pyroids) (1.5.7)
Installing collected packages: pyroids
Successfully installed pyroids-0.9.0
osx$
```

Fig. 2.7 Ejemplo de uso de pip para instalar el paquete ‘pyroids’ y sus dependencias.

El otro manejador de paquetes ampliamente utilizado es [conda](#), que utiliza el repositorio de paquetes de Anaconda <https://repo.anaconda.com/>: si usted instaló Python en su computador utilizando Anaconda, probablemente sea buena idea que utilice [conda](#) para manejar la instalación de paquetes en su máquina.

2.2.5.2. Notebooks

En el mundo Python es frecuente que en lugar de utilizar un IDE, donde cada programa se construye en uno o varios archivos python, se utilicen unas estructuras llamadas notebooks. En síntesis, un notebook es un archivo que se lee y se edita a través de una aplicación web y que puede mezclar texto con fragmentos de código que se pueden ejecutar dentro del notebook. La [figura 2.8](#) muestra una captura de pantalla de un notebook en uso: tiene una *celda* en la que hay un texto con formato y luego tiene una *celda* con un fragmento de un programa y con el resultado de su ejecución abajo.

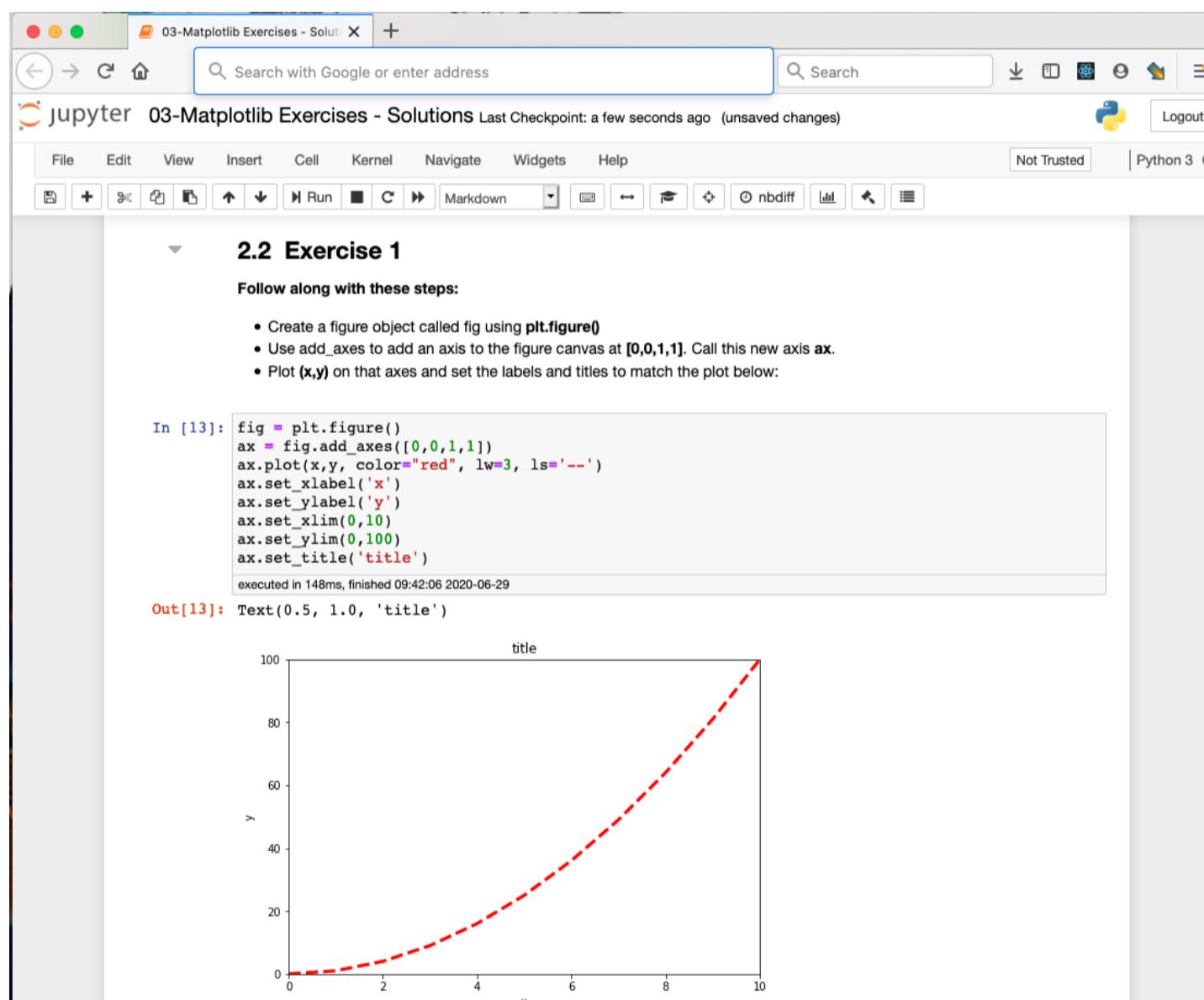


Fig. 2.8 Captura de pantalla de un *notebook* con Jupyter

Si quiere saber más sobre notebooks puede visitar la página del proyecto Jupyter, los cuales desarrollan la principal herramienta para soportar notebooks: <https://jupyter.org/>. Sin embargo, su instalación y uso puede no ser tan fácil para un principiante.

- [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.
- [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.

[On this page](#)[2.3.1. Números enteros \(int\)](#)[2.3.2. Números decimales \(float\)](#)[2.3.3. Cadenas de caracteres \(str\)](#)[2.3.4. Conversiones entre tipos de datos](#)[2.3.5. Ejercicios](#)

Versión borrador / preliminar

[Print to PDF ▶](#)

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.

2.3. Valores y tipos de datos

Objetivo de la sección

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 `>>>`.

2.3.1. 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 [1].

2.3.2. 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.

Importante

Use siempre un punto y no una coma para separar la parte entera de la parte decimal en un `float`.

2.3.3. 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 quisieramos 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 carácter ' 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 [2].

```
>>> """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 escrita 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 carácter 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'
```

Importante

Para representar cambios de línea dentro de una cadena de caracteres debe usarse la combinación `\n`.

2.3.4. 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'`.

Cuidado

Si usted intenta convertir valores de forma equivocada obtendrá un error y su programa posiblemente se detendrá. En general, usted debería verificar los valores para asegurarse que no vaya a tener problemas antes de intentar hacer una conversión de tipos. Más adelante veremos cómo se puede lograr esto.

2.3.5. Ejercicios

1. Abra el intérprete de Python e intente convertir a entero las siguientes cadenas. ¿Qué resultado obtiene?

- '1'
- '1.1'
- '-3'
- 'a'

2. Abra el intérprete de Python e intente convertir a entero los siguientes números. ¿Qué resultado obtiene?

- 3
- 2.86
- 2,86
- 3.4
- 3.4

[1] 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).

[2] El texto fue tomado de <http://shakespeareshaiku.blogspot.com/> y está basado en el Soneto 141 de William Shakespeare.

By Mario Sánchez
© Copyright Agosto de 2020.

[Departamento de Ingeniería de Sistemas y Computación](#) - Universidad de los Andes

[On this page](#)[2.4.1. Analizando la primera instrucción](#)[2.4.2. Operadores y expresiones](#)[2.4.3. Instrucciones con variables](#)[2.4.4. Paréntesis y tipos de datos](#)[2.4.5. Otro tipo de instrucciones](#)[2.4.6. Comentarios en Python](#)[2.4.7. Más operadores en Python](#)[2.4.7.1. Operadores para números](#)[2.4.7.2. Operadores para cadenas de caracteres](#)[2.4.7.3. Operadores con acumulación](#)[2.4.7.4. Expresiones con múltiples tipos de datos](#)[2.4.8. Ejercicios](#)[2.4.9. Más allá de Python](#)

Versión borrador / preliminar

[Print to PDF](#)

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.

2.4. Variables, expresiones y operadores

Objetivo de la sección

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, comenzando 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!

2.4.1. Analizando la primera instrucción

Analicemos ahora la primera instrucción del programa:

```
v1 = 5
```

Esta instrucción se puede separar en dos partes divididas por el carácter `=`. En la parte derecha, encontramos sólo el carácter `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, `v1` representa 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 carácter `=` que separa la *variable* que se encuentra a la izquierda del *valor* que se encuentra a la derecha. En Python, el carácter `=` 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 carácter 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` [2].

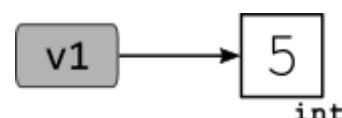


Fig. 2.9 Representación gráfica de la variable `v1` que contiene el valor `5` que es de tipo `int`

⚠️ 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`

2.4.2. 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`.

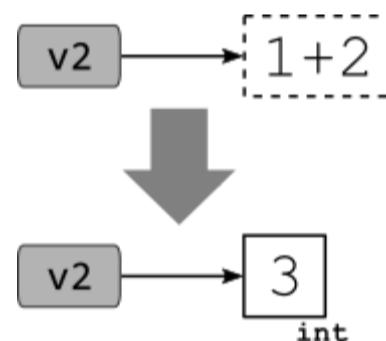


Fig. 2.10 Representación gráfica de la variable `v2` y de cómo se calcula su valor.

2.4.3. 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` [3].

```
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.

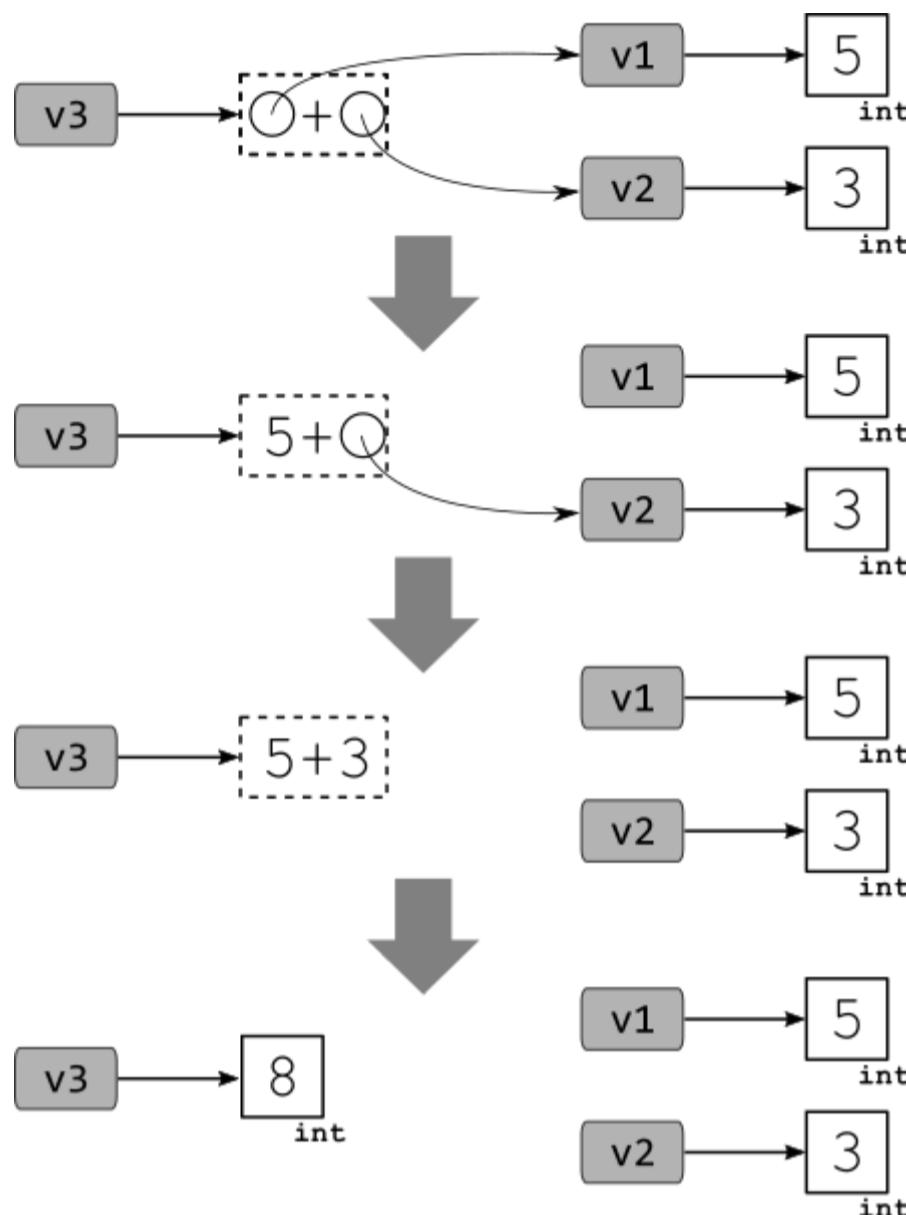


Fig. 2.11 Representación gráfica de la variable v3

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
```

2.4.4. 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, Python sólo podrá calcular el resultado de la división después de haber calculado el valor de la suma.

En este programa, Python empezará a calcular los valores que se están dividiendo, empezando por el de la izquierda. Es decir que buscará el valor de la variable `v3` y la instrucción se reescribirá como:

```
v4 = 8 / (4 + 1)
```

Debido a que el divisor sigue siendo una expresión, Python evaluará su valor y reescribirá los que estaba dentro de los paréntesis. La nueva estructura de la instrucción será:

```
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`.

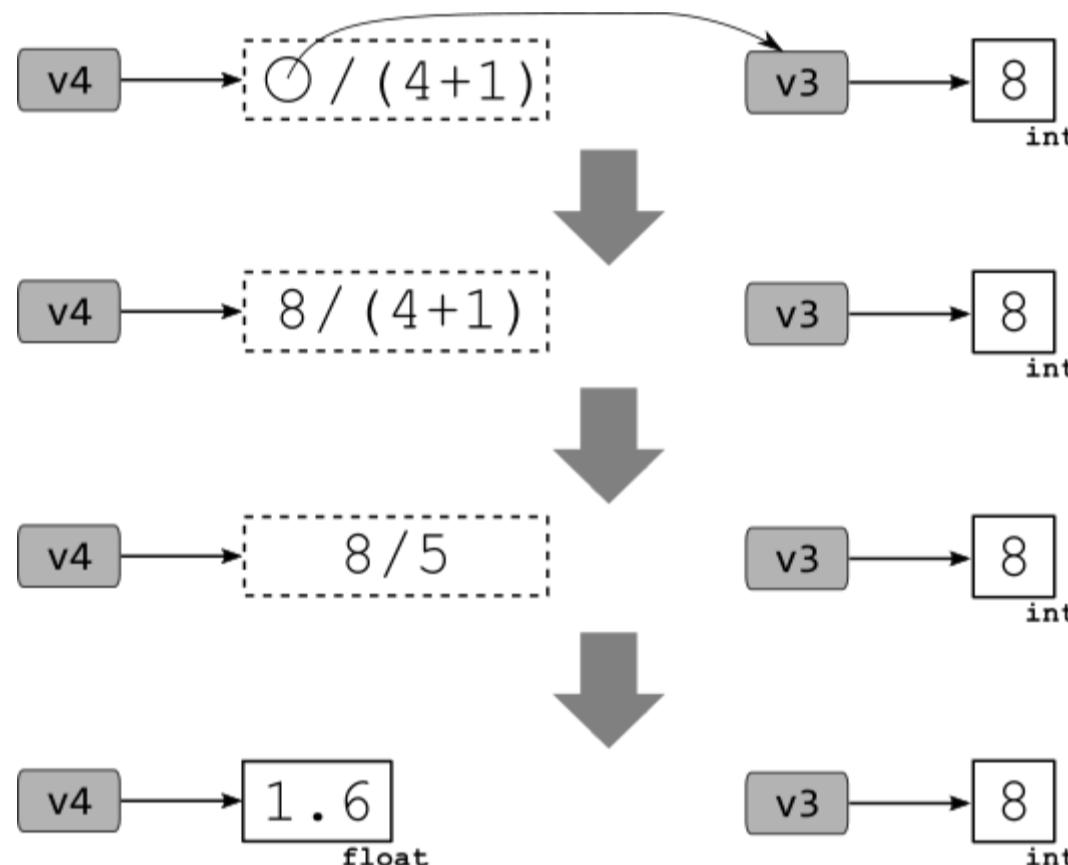


Fig. 2.12 Representación gráfica de la variable `v4`

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'>
```

2.4.5. Otro tipo de instrucciones

Si revisamos con cuidado las 4 instrucciones que hemos analizado, nos daremos cuenta que 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: 1.6
```

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.

2.4.6. 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 carácter #: todos los caracteres que se encuentren a la derecha de este carácter serán ignorados por el intérprete.

En el caso de nuestro ejemplo, 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.

💡 Tip

Si tiene dudas sobre si debería incluir un o no, entonces inclúyalo. En la mayoría de los casos siempre será mejor tener *más* comentarios que *menos* comentarios.

2.4.7. 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.

2.4.7.1. 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	3	10 // 3	3
Módulo	%	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 `3.333333333333335`, 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) `3.0`.

(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`.

2.4.7.1.1. El operador módulo (%)

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 [4].

La operación módulo y la división entera se complementan. Cuando se está aprendiendo a dividir, es común que el resultado de una división se exprese como un valor y su residuo. Así, `27 dividido 10` es igual a `2` con un residuo de `7`. En Python, esto se puede calcular usando los operadores `//` y `%`:

```
>>> entero = 27 // 10
>>> residuo = 27 % 10
>>> print(entero, residuo)
2 7
```

La operación módulo es muy utilizada también 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.

2.4.7.2. 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.

2.4.7.3. Operadores con acumulación

En Python existen también versiones alternativas de algunos de los operadores presentados que permiten acumular los valores. Veamos primero un ejemplo de esto:

```
>>> a = 1
>>> a += 2
>>> a += 5
>>> print(a)
8
```

En la primera instrucción del programa anterior estamos creando una nueva variable con el valor `1`. Luego, le estamos sumando `2` a esa variable y estamos guardando el valor en la misma variable. Finalmente, le sumamos `5` al valor almacenado, y volvemos a guardar el resultado en la variable `a`. El programa anterior es equivalente al siguiente:

```
>>> a = 1
>>> a = a + 2
>>> a = a + 5
>>> print(a)
8
```

Como se puede ver, la única diferencia significativa es que el primer programa es un poco más corto. Otros operadores con acumulación disponibles incluyen `-=`, `*=`, `/=`, `//=`, `%=`, `**=`. Note que cuando el operador `+=` se aplica sobre cadenas de caracteres, la operación realizada es la concatenación.

2.4.7.4. 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 eliminan los posibles problemas.

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

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

2.4.8. 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 [5]
- 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)
```

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

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

5. ¿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
```

2.4.9. 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* [6]. 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.

[2] 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.

[3]

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.

- [4] 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.
- [5] Tenga cuidado con aplicando el operador de división entera sobre números negativos: si uno de los dos operandos es negativo, el resultado será el mismo de aplicar la función `piso` al resultado de la división (es decir, se retorna el mayor entero que sea menor que el resultado).
- [6] 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.

By Mario Sánchez
© Copyright Agosto de 2020.

[Departamento de Ingeniería de Sistemas y Computación](#) - Universidad de los Andes

2.5. Funciones

[Print to PDF](#)
[On this page](#)
[2.5.1. El concepto de función](#)
[2.5.2. Funciones en Python](#)
[2.5.2.1. Funciones de conversión: int, float y str](#)
[2.5.2.2. Funciones numéricas: abs, round, min, max, pow](#)
[2.5.2.3. Funciones de entrada y salida: print, input](#)
[2.5.2.4. Funciones sobre caracteres: chr, ord](#)
[2.5.3. Ejercicios](#)
[2.5.4. Más allá de Python](#)

⚠ 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.

💡 Objetivo de la sección

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.

2.5.1. 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* cuyo valor depende de dos parámetros llamados *x* y *y*: cuando se evalua 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 independiente 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[1] 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:

$$\text{potencia}(\text{base}, \text{exponente}) = \text{base}^{\text{exponente}}$$

2.5.2. 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 *evalúe* 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.
```

2.5.2.1. Funciones de conversión: `int`, `float` y `str`

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.

2.5.2.1.1. `int(x)`

La función[2] `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[3].

- 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 evalua 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`.

2.5.2.1.2. float(x)

La función[2] `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 evalua 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`.

2.5.2.1.3. str(x)

La función `str` es una función tremadamente 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+2j)`, valor `'3.14'`

2.5.2.2. Funciones numéricas: abs, round, min, max, pow

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

2.5.2.2.1. 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'
```

2.5.2.2. 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
```

2.5.2.2.3. round(n)

La función `round` también se puede utilizar con un sólo parámetro (el número `n`), en cuyo caso el resultado será un valor entero.

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

```
>>> round(3.14159)
3
>>> round(-3.14159)
-3
>>> round(4.89)
5
>>> round(-4.89)
-5
```

2.5.2.2.4. 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
```

2.5.2.2.5. 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
```

2.5.2.3. Funciones de entrada y salida: print, input

A continuación describiremos dos funciones que son tremadamente 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.

2.5.2.3.1. 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 [4].

A continuación se muestra un ejemplo que 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.

2.5.2.3.2. 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.

2.5.2.4. Funciones sobre caracteres: chr, ord

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

2.5.2.4.1. 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 127 a un conjunto de caracteres. Por ejemplo, la tabla dice que al carácter 'A' le corresponde el número 65, que al carácter 'a' le corresponde el 97, al carácter '7' le corresponde el 55, y al carácter '!' 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 carácter 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 carácter tiene asignado el número 10. Otros caracteres de control importantes son el número 9 que representa una tabulación ('\t', el carácter que se introduce cuando se presiona la tecla TAB en un teclado) y el carácter 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' [5].

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 carácter número 97 en UNICODE también es el carácter 'a'.

2.5.2.4.2. 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 carácter 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 carácter 'a'. La función `ord` tiene el objetivo opuesto: dado un carácter, indica cuál 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('ə')
9787
>>> chr(9787)
'ə'
>>> chr(22223)
'囂'
```

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

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

2.5.3. 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 de 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 dinero en pesos, una tasa de interés (un número decimal mayor a 0) 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 $C \times (1 + x/100)^n$ pesos. Pruebe su programa sabiendo que una cantidad de 10,000 pesos al 4.5 % de interés anual se convierte en 24117.14 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 x_1 , x_2 y x_3 . El programa luego debe rotar las variables de forma que al final x_2 tenga el valor inicial de x_1 , x_3 el de x_2 y x_1 el de x_3 .

2.5.4. 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.

- [1] Usamos acá el término *parámetro* para evitar confusiones que podrían aparecer si usaramos el término *variable*.
- [2][1:2] 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.
- [3] La función `int` también puede invocarse con un parámetro nombrado `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.
- [4] Más adelante estudiaremos usos más avanzados de la función que permiten, por ejemplo, separar los elementos usando un carácter diferente.
- [5] Otro carácter interesante es el que tiene el número 7 y que es llamado “beep”: cuando se *imprime* este carácter en la consola, el computador debería emitir un anuncio sonoro. En sistemas operativos modernos, el sonido utilizado es el sonido de alerta configurado.

By Mario Sánchez
© Copyright Agosto de 2020.

[Departamento de Ingeniería de Sistemas y Computación](#) - Universidad de los Andes

2.6. Definición de funciones

[Print to PDF ▶](#)

⚠ 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.

ℹ️ Objetivo de la sección

El objetivo de esta sección es seguir explorando el concepto de función y aprender a definir nuevas funciones.

2.6.1. Antes de empezar: módulos y 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.

2.6.2. Un ejemplo completo: el programa de la casa

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 cuenta que cuando se habla de “una casa” se hace referencia a un dibujo como el siguiente:

☰ On this page

- [2.6.1. Antes de empezar: módulos y archivos .py](#)
- [2.6.2. Un ejemplo completo: el programa de la casa](#)
- [2.6.3. Definición de funciones en Python](#)
 - [2.6.3.1. La signatura de una función](#)
 - [2.6.3.2. El cuerpo de una función](#)
- [2.6.4. Definir vs. Invocar](#)
 - [2.6.4.1. Definición de la función](#)
 - [2.6.4.2. Invocación de la función](#)
 - [2.6.4.3. Ejecución de un programa](#)
- [2.6.5. Funciones sin parámetro o sin retorno](#)
 - [2.6.5.1. Funciones sin parámetros](#)
 - [2.6.5.2. Funciones sin retorno](#)
- [2.6.6. Ejercicios](#)
- [2.6.7. Más allá de Python](#)

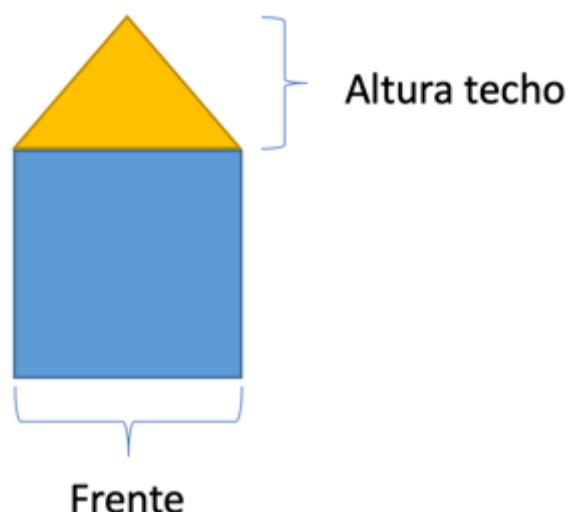


Fig. 2.13 Representación básica de una casa.

```
# 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.

2.6.3. 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.

2.6.3.1. La firma de una función

La firma 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 [\[1\]](#) 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 [\[2\]](#).

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 paréntesis redondos.

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 función 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`).

La [figura 2.14](#) presenta de manera gráfica esta misma información: la función `area_cuadrado` requiere una entrada de tipo `int` y produce un resultado también de tipo `int`. Más adelante exploraremos qué es lo que pasa dentro de la función que convierte la entrada `lado` en un resultado.

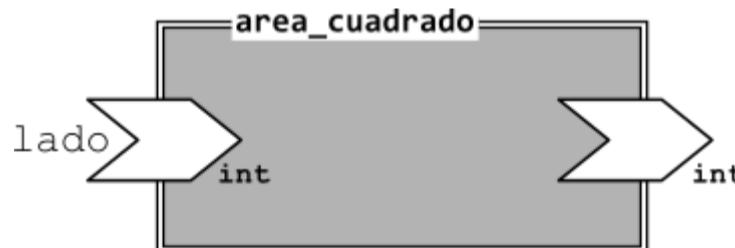


Fig. 2.14 Representación gráfica de la signatura de la función `area_cuadrado`

⚠️ Cuidado

La única forma de introducir información dentro de una función debería ser a través de los parámetros. Aunque desde el cuerpo de una función pueden leerse variables que no aparezcan en los parámetros, es una pésima práctica que usualmente lleva a errores difíciles de encontrar y corregir [\[3\]](#).

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.

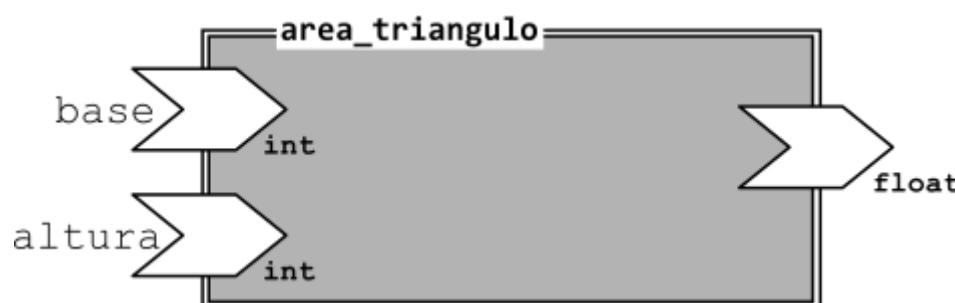


Fig. 2.15 Representación gráfica de la firma de la función `area_triangulo`

Actividades:

1. Escriba la firma de una función que sirva para calcular el área de un círculo dado su radio.
2. Escriba la firma 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.
3. Siguiendo los ejemplos anteriores, haga una representación gráfica de su nueva función.

2.6.3.2. 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: es 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 espacios 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. En la siguiente sección 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 a lo especificado en la firma de la función.

Ahora bien, antes de retornar un valor, es necesario que Python evalúe 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 firma de la función. La expresión `lado * lado` está haciendo entonces referencia al parámetro que le pasen a la función cada vez que la invoquen.

La [figura 2.16](#) representa esto mismo gráficamente. Note que ahora sí incluimos elementos dentro de la función (ya no es una caja negra) y que el valor que entra como parámetro `lado` se usa dentro de la expresión `lado * lado`.

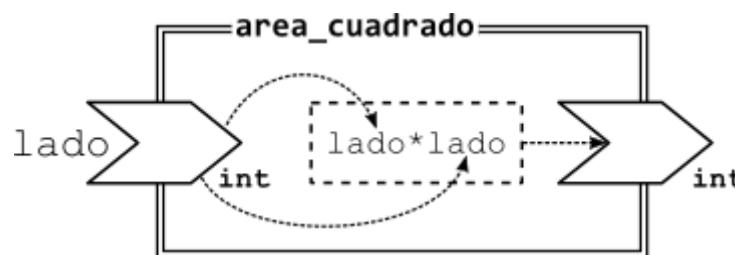


Fig. 2.16 Representación gráfica de la función `area_cuadrado`, incluyendo su implementación

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.

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?

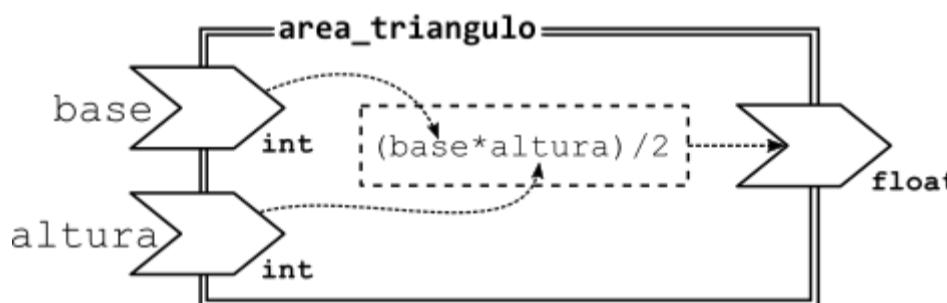


Fig. 2.17 Representación gráfica de la función `area_triangulo`, incluyendo su implementación.

Actividades:

1. Escriba el cuerpo de la función para calcular el área de un círculo dado su radio. Utilice la signatura que definió en una de las actividades anteriores.
2. Escriba el cuerpo de la función para calcular la velocidad final de un objeto. Utilice la signatura que definió en una de las actividades anteriores.
3. Siguiendo los ejemplos anteriores, haga una representación gráfica de la función para calcular la velocidad final de un objeto.

2.6.4. 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")
```

2.6.4.1. 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 firma 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 firma de la función `area_casa` sólo tiene una pequeña diferencia: utiliza diferentes tipos de datos (sus parámetros son `int` y su resultado es `float`).

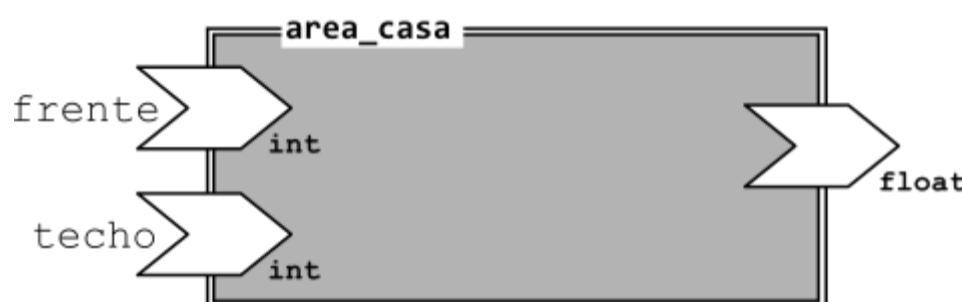


Fig. 2.18 Representación gráfica de la firma de la función `area_casa`.

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`.

⚠️ Cuidado

Las variables que se definen dentro de una función sólo pueden leerse y escribirse dentro de esa función. Si intentamos leer esa variable por fuera de la función se producirá un error similar al siguiente:

```
NameError: name 'variable' is not defined
```

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.

La [figura 2.19](#) muestra todo esto de manera gráfica: la función `area_casa` requiere dos entradas (`frente` y `techo`), las cuales se usan en invocaciones a las funciones `area_cuadrado` y `area_triangulo`. Los resultados de estas invocaciones se guardan en dos variables (`cuadrado` y `triangulo`), que finalmente se suman para producir el resultado de la función

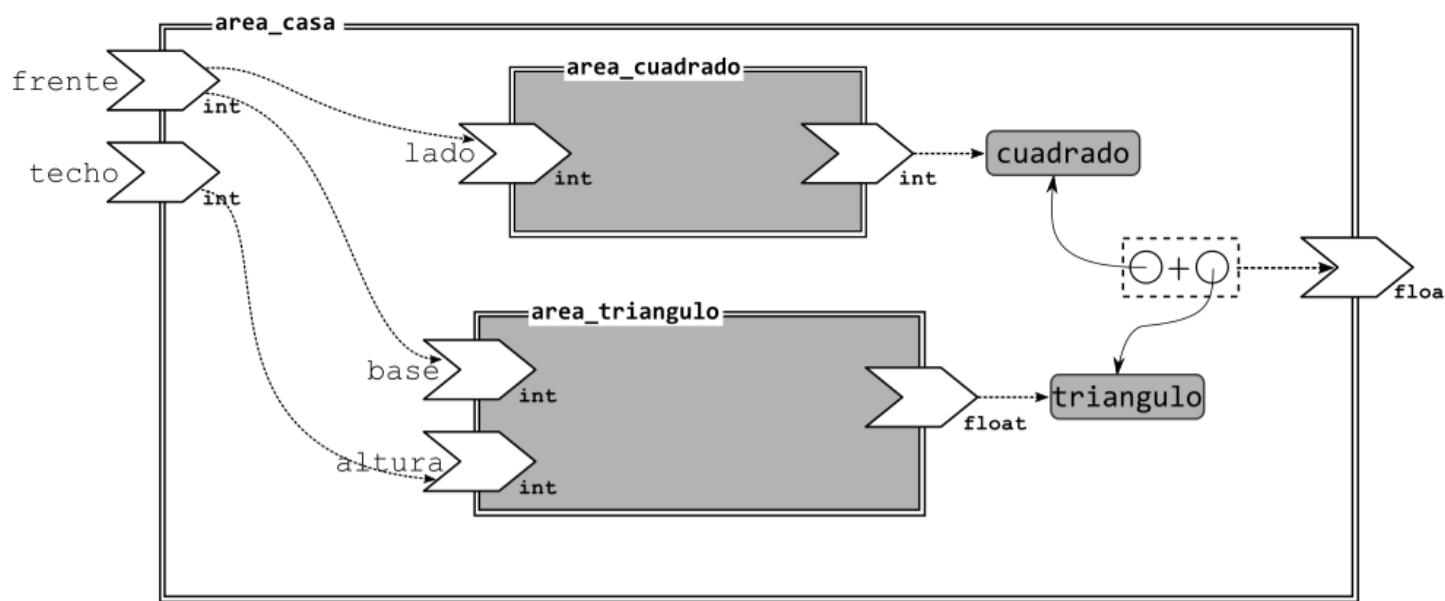


Fig. 2.19 Representación gráfica de la función `area_casa`.

! Importante

La [figura 2.19](#) muestra a las funciones `area_cuadrado` y `area_triangulo` como caja negra para ilustrar un concepto muy importante: cuando se invoca una función, lo único que interesa son su nombre, sus entradas y su salida. Los detalles sobre su implementación (lo que pasa dentro de la caja negra) no nos importa.

💡 Tip

El nombre que se le da a un parámetro en la firma 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.

2.6.4.2. 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`.

La [figura 2.20](#) representa gráficamente esta situación: las dos variables se usan como parámetros para la función `area_casa` y el resultado de la invocación se almacena en la variable `resultado`. La función aparece representada como una caja negra porque en el momento de la invocación, sólo nos importan los parámetros de entrada y su resultado.

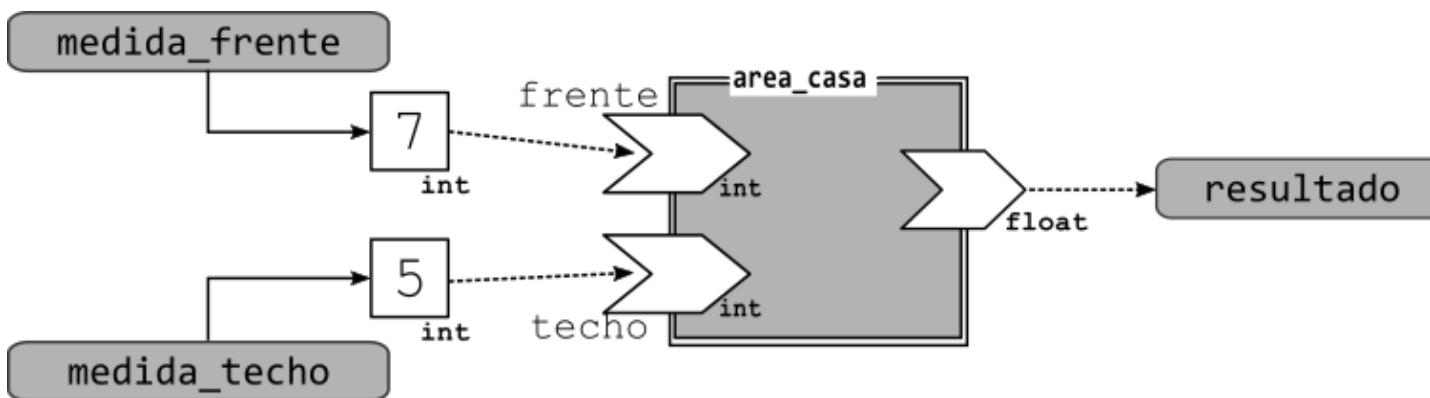


Fig. 2.20 Representación gráfica del programa, incluyendo la invocación a la función `area_casa`.

Como nosotros 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.

2.6.4.3. 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:

```

$ python saludar.py
¿Cuál es su nombre? Alicia
Hola Alicia!
$ 
  
```

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.

2.6.5. 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 afirmativa: 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.

2.6.5.1. 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 firma, 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.

2.6.5.2. 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.

2.6.6. 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.

2.6.7. 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 firmas. 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 equivalentes en Python.

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/>

[1] 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.

[2] 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`.

[3] Esto es algo que se puede hacer en Python pero no suele ser posible en otros lenguajes, precisamente porque es muy mala idea.

2.7. Estilo de programación

⚠ 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.

ℹ Objetivo de la sección

El objetivo de esta sección es introducir algunos elementos que no son obligatorios al programar pero que hacen el código mucho más comprensible y fácil de mantener.

En las secciones anteriores hemos empezado a estudiar la *sintaxis* de Python, es decir las reglas que define el lenguaje y que deben respetarse en todos los programas que se escriban con él: si las reglas sintácticas no se cumplen, el intérprete de Python no podrá ejecutarlo. Print to PDF Sin embargo, la sintaxis no lo es todo en un lenguaje de programación. También es importante pensar en el *estilo*.

Si bien las reglas de Python parecen muy estrictas, hay infinidad de formas diferentes en las que se puede escribir el mismo programa. Esto incluye desde cambios menores, como incluir líneas en blanco o comentarios adicionales, hasta cambiar la descomposición en funciones, pasando por cambios en los nombres de las variables y de las funciones. Es decir que dos personas que escriban el mismo programa Python pueden terminar escribiendo programas muy diferentes sólo por utilizar un *estilo* diferente.

El problema con esta gran libertad es que las decisiones que tomemos con respecto al *estilo* del código que escribamos hoy posiblemente nos van a acompañar durante mucho tiempo. La mayoría de los programas se escriben en un tiempo relativamente corto (días o semanas) y luego se utilizan, se corrigen y se actualizan durante meses o años. Si hoy tomamos decisiones que parecen fáciles (como utilizar nombres de variables cortísimos o no descomponer una función muy complicada), es posible que más adelante sea más difícil arreglar un problema porque no podremos entender con facilidad nuestro propio código.

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

—Martin Fowler

Para ilustrar este punto, revisemos el siguiente programa basado en las funciones que usamos en la sección anterior:

☰ On this page

[2.7.1. Nombramiento de variables y funciones](#)

[2.7.1.1. Estándares](#)

[2.7.2. Documentación de funciones](#)

[2.7.2.1. Ejercicios](#)

[2.7.3. Descomposición de funciones](#)

[2.7.4. Complejidad de las instrucciones](#)

[2.7.5. Tipado de funciones y parámetros](#)

[2.7.6. Comentarios](#)

[2.7.7. Ejercicios](#)

[2.7.8. Más allá de Python](#)

```

1 def area_cuadrado(lado: int)-> int:
2     """ Calcula el área de un cuadrado dada La medida de su Lado
3     Parámetros:
4         lado (int): La medida del cuadrado
5     Retorno:
6         (int): El valor del área del cuadrado. Es siempre un número entero.
7     """
8     return lado * lado
9
10
11 def area_triangulo(base: int, altura: int)-> float:
12     """ Calcula el área de un triángulo.
13     Parámetros:
14         base (int): La medida de la base del triángulo.
15         altura (int): La medida de la altura del triángulo.
16     Retorno:
17         (float): El valor del área del triángulo. Es siempre un número decimal.
18     """
19     return (base * altura) / 2
20
21
22 def area_casa(frente: int, techo: int)-> float:
23     """ Calcula el área del dibujo de una casa que se forma con un cuadrado
24         y un triángulo encima (el techo).
25         El frente de la casa será igual al lado del cuadrado y a la base del triángulo.
26         La altura del techo será la altura del triángulo.
27     Parámetros:
28         frente (int): La medida del frente de la casa.
29         techo (int): La medida de la altura del techo de la casa.
30     Retorno:
31         (float): El valor del área del dibujo de la casa.
32     """
33     cuadrado = area_cuadrado(frente)
34     triangulo = area_triangulo(frente, techo)
35     return cuadrado + triangulo
36
37 print(area_casa(10, 5))

```

Ahora revisemos un segundo programa:

```

1 def f(b, c):
2     v = (b * b) + (b * c)/2
3     return v
4
5 print(f(10, 5))

```

Aunque a primera vista no es evidente, los dos programas son equivalentes en el sentido de que al ejecutarlos el resultado será el mismo: imprimirán el valor **125.0** en la consola. Evidentemente el segundo programa es mucho más corto que el primero, pero esto no necesariamente es una ventaja. En este caso es difícil entender que la función **f** sirve para calcular el área del dibujo de la casa y requiere como parámetro las medidas del frente y del techo, en ese orden.

Podemos decir que las diferencias entre los dos programas se reducen a los siguientes 5 aspectos:

1. Utilizar nombres claros para las variables, las funciones y los parámetros.
2. Documentar el objetivo de cada función
3. Descomponer las funciones para que cumplan objetivos precisos
4. Complejidad de las instrucciones
5. Indicar los tipos de los parámetros y retornos de las funciones

A continuación revisaremos cada uno de estos puntos con un poco más de detalle.

2.7.1. Nombramiento de variables y funciones

Uno de los factores que más incide en la facilidad para comprender un programa es la selección de nombres para variables y funciones: si el nombre que utilizamos para una variable o una función es bueno, no tendremos que pensar mucho para recordar qué rol tiene dentro del programa y podremos concentrarnos en los aspectos importantes.

En el ejemplo que presentamos antes podemos ver esto claramente:

```

1 def area_triangulo(base: int, altura: int)-> float:
2     return (base * altura) / 2
3
4 def g(b, c):
5     return (b * c)/2

```

Las dos funciones anteriores hacen los mismos cálculos, pero la primera es claramente más explícita: no tenemos que hacer un gran esfuerzo para descubrir cuál es su objetivo.

Use buenos nombres de variables

Para las variables, utilice nombres que indiquen con claridad qué es lo que va a guardar dentro de ellas. Evite nombres muy cortos a menos que no haya ninguna ambigüedad posible.

Use buenos nombres de funciones

Para las funciones, utilice nombres que indiquen qué es lo que hará la función. Incluya verbos en los nombres de funciones (ej. `guardar_resultado`), a menos que se pueda sobreentender con facilidad (ej. `calcular_area_triangulo` vs. `area_triangulo`).

2.7.1.1. Estándares

Más allá de los nombres y de lo que significan, en cada lenguaje también hay estándares para el uso de mayúsculas y minúsculas y la separación de palabras en los nombres de variables y funciones. Estos estándares pueden parecer arbitrarios (¡y lo son!) pero es importante respetarlos porque seguirlos consistentemente hace mucho más sencilla la lectura del código.

En Python, las reglas más importantes en este sentido son las siguientes:

1. Usar *snake_case*. Esto significa que las palabras de un identificador deberían separarse usando el carácter `'_'`. Por ejemplo, en Python se prefiere usar `calcular_area_triangulo` mientras que en Java se usaría `calcularAreaTriangulo`.
2. Usar *minúsculas* para los identificadores. Tanto funciones como variables y parámetros deberían nombrarse usando minúsculas.
3. Usar *Mayúscula Inicial* para los nombres de clases [1].
4. Usar *MAYÚSCULAS SOSTENIDAS* para las constantes. Aunque estrictamente hablando en Python no existe el concepto de constante, se suelen usar mayúsculas sostenidas para indicar que el valor de una variable no debería cambiar su valor. Por ejemplo, `ROJO` o `IVA`. Desafortunadamente una de las constantes más útiles, `math.pi`, no sigue este estándar.

Use el alfabeto inglés

Aunque en Python es posible utilizar en los identificadores caracteres que existen en el español pero no existen en inglés, como la `ñ` y las vocales acentuadas, es recomendable evitarlo para evitar problemas de codificación. Esto es especialmente importante si se va a usar el mismo código en máquinas Windows, Linux y Mac.

2.7.2. Documentación de funciones

Un segundo aspecto para facilitar el uso de nuestro código es documentar las funciones con un comentario que le sirva a potenciales usuarios o a nosotros mismos. De esta forma no será necesario estudiar con detenimiento el cuerpo de la función para saber qué hace.

Para cada función usualmente queremos saber 4 cosas:

1. cuál es su objetivo
2. cómo se debe usar
3. qué pasará cuando se use
4. cómo deben usarse y qué representan los parámetros

En Python el comentario con múltiples líneas que se encuentre justo después de la firma de una función es considerado la documentación de la función. Esto usualmente se conoce como el `docstring` de una función y tiene una característica muy importante: cuando busquemos ayuda sobre una función, usando la función nativa `help`,

recibiremos el [docstring](#). Veamos un ejemplo en el que primero definiremos una nueva función y especificaremos su [docstring](#).

```

1 def area_triangulo(base: int, altura: int) -> float:
2     """ Calcula el área de un triángulo a partir de su base y su altura.
3         Tanto la base como la altura deben ser números enteros.
4         El resultado es un número decimal aunque los parámetros sean enteros.
5     """
6     return (base * altura) / 2

```

Si después de definir nuestra función invitamos la función nativa [help](#) usando nuestra función como parámetro, obtendremos la documentación que especificamos.

```

>>> help(area_triangulo)
Help on function area_triangulo in module __main__:

area_triangulo(base: int, altura: int) -> float
    Calcula el área de un triángulo a partir de su base y su altura.
    Tanto la base como la altura deben ser números enteros.
    El resultado es un número decimal aunque los parámetros sean enteros.

```

Ahora bien, a diferencia de otros lenguajes Python no especifica cómo deben describirse los detalles de una función: sólo nos da el espacio para que escribamos la documentación y nos da total libertad para que nosotros decidamos qué aspectos queremos documentar. Es nuestra responsabilidad decidir qué incluir y asegurarnos de que la documentación sea suficiente para que alguien más pueda usar nuestra función. También es nuestra responsabilidad definir cómo vamos a organizar la información para que esté organizada y sea fácil de encontrar y utilizar.

Toda esta libertad que da el lenguaje ha llevado a que existan varios estándares para documentar las funciones sin que ninguno sea claramente superior a los otros. Aunque le recomendamos que más adelante escoja uno de los estándares, por ahora le recomendamos utilizar la versión simplificada que se ilustra en el siguiente ejemplo:

```

1     """ Calcula el área de un triángulo a partir de su base y su altura.
2     Parámetros:
3         base (int): La medida de la base del triángulo.
4             Debe ser un número estrictamente positivo (mayor o igual a 1).
5         altura (int): La medida de la altura del triángulo.
6             Debe ser un número estrictamente positivo (mayor o igual a 1).
7     Retorno:
8         (float): El valor del área del triángulo. Es siempre un número decimal.
9     """

```

Esta documentación incluye los siguientes elementos:

1. Descripción de la función. Acá explicamos cuál es el objetivo de la función para que sea fácil saber si es la función que necesitamos usar. Además, si la función es muy complicada, explicamos qué es lo que hace la función por dentro. Esta descripción puede ocupar múltiples líneas: le recomendamos que no use líneas muy largas y que intente que el texto quede bien alineado a la izquierda.
2. Parámetros. Si la función tiene parámetros, especificamos el nombre y el tipo de cada uno, seguidos de una descripción. La idea es que quien vaya a utilizar la función se entere de qué representa el parámetro y de todas las reglas que deberían aplicarse.
3. Retorno. Acá explicamos qué es lo que retorna la función.

Para una función tan sencilla como la del ejemplo, puede parecer que esta descripción tan grande es exagerada.

Pronto estaremos trabajando con funciones mucho más complicadas en las que será muy importante que escribamos una documentación muy completa para que no nos confundamos nosotros mismos o confundamos a las personas con las que estemos trabajando.

Documente sus funciones

Documente siempre sus funciones utilizando un formato consistente que incluya una descripción general y la explicación detallada de los parámetros y el retorno.

2.7.2.1. Ejercicios

1. Use la función [help](#) para consultar la documentación de algunas de las funciones nativas que ya ha estudiado.

2. Escriba la documentación de alguna función que haya desarrollado en un ejercicio previo. Revise en el intérprete de Python que pueda leer la documentación de la función usando la función `help`.

2.7.3. Descomposición de funciones

Aunque no se puede generalizar, en lo posible deberíamos tener funciones sencillas que se compongan poco a poco para formar funciones más complicadas. Esto es preferible a tener funciones extremadamente complicadas que se tengan que leer con muchísima atención: al leer el código de una función debería ser claro cuál es su objetivo principal y cómo lo está logrando.

Para lograr una buena descomposición es necesario primero hacer abstracción de las funciones, separando la firma de la implementación. Es decir, debemos pensar en qué se quiere lograr con una función independientemente de cómo se vaya a implementar. El proceso se debe repetir identificando funciones cada vez más sencillas que sirvan para explicar cómo se resuelven las funciones más grandes, pero sin entrar en detalles, hasta que lleguemos a funciones triviales. En una sección posterior estudiaremos en mucho más detalle este proceso que se conoce como *refinamiento a pasos*.

Simplifique sus funciones

Intente tener funciones que tengan un único objetivo y que sean fáciles de explicar. Si usted descubre que el objetivo o la implementación de una función son muy complicados de explicar, posiblemente sea una señal de que debe descomponerla en funciones más pequeñas.

El siguiente motivo por el cual tiene sentido descomponer las funciones es para evitar la repetición de código. En general, tener código repetido es mala idea porque aumenta la posibilidad de tener errores y porque, en caso de querer corregir un error, será necesario hacerlo en muchos lugares.

Use funciones para evitar repeticiones

Si está repitiendo el mismo código en varios lugares, considere construir una función que encapsule esa funcionalidad y que pueda llamar en todos los lugares donde lo requiera.

2.7.4. Complejidad de las instrucciones

Una razón por la cual muchas veces el código es mucho más difícil de leer y entender de lo necesario es porque se hacen muchas acciones dentro de la misma instrucción. A manera de ejemplo a continuación presentamos dos funciones equivalentes que calculan el área de un polígono regular a partir de la longitud de un lado y de la cantidad de lados:

```
1 import math
2
3 def area_poligono(lado: float, num_lados: int) -> float:
4     return (num_lados * lado**2) / (4 * math.tan(math.pi / num_lados))
```

```
1 import math
2
3 def area_poligono2(lado: float, num_lados: int) -> float:
4     angulo_interno_radianes = math.pi / num_lados
5     numerador = num_lados * lado**2
6     denominador = 4 * math.tan(angulo_interno_radianes)
7     return numerador / denominador
```

Aunque el primer ejemplo no es extremadamente complicado, la única instrucción que tiene es mucho más complicada que cualquiera de las instrucciones del segundo ejemplo. Esto significa que el segundo ejemplo será más fácil de leer y probablemente fue más fácil de construir que el primero.

Simplifique las instrucciones

Escriba instrucciones que sean lo más sencillas posibles.

Idealmente, cada línea de código debería hacer una sola cosa.

2.7.5. Tipado de funciones y parámetros

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* 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.

Utilice *type-hints*

Utilice los *type-hints* para todos los parámetros y los retornos de las funciones. No sólo harán que su código sea más legible y fácil de usar, sino que además lo prepararán a usted para utilizar otros lenguajes.

2.7.6. Comentarios

Por último, hay un aspecto adicional que es muy sencillo pero tiende a mejorar la calidad del código: introducir comentarios dentro de las instrucciones. En los ejemplos que introducimos al principio de la sección eso no se estaba haciendo porque las funciones utilizadas eran muy sencillas, pero en funciones como las que estudiaremos a partir de la próxima sección esto será mucho más importante.

En general deberían incluirse comentarios dentro del código para explicar el funcionamiento de bloques de código que sean particularmente complicados. No existe ningún estándar sobre esos comentarios, pero a continuación le damos algunas recomendaciones:

1. No exagere con los comentarios. Así como la falta de comentarios es grave, el exceso de comentarios puede terminar en código muy difícil de leer.
2. Escriba comentarios que expliquen lo que hacen fragmentos significativos de código y/o su justificación, en lugar de hacer una traducción de Python a español de las instrucciones realizadas.
3. Identifique las instrucciones particularmente complicadas y documéntelas.
4. En funciones medianas o largas que no pueda o no quiera descomponer, enumere las grandes etapas usando comentarios.

2.7.7. Ejercicios

1. Revise detenidamente la siguiente función para descubrir su objetivo. Reescríbalo aplicando las recomendaciones que se estudiaron en esta sección.

```
def vc(r, a):
    b = 3.14159 * (r**2)
    return round(b * a, 2)
```

1. Revise detenidamente la siguiente función para descubrir su objetivo. Reescríbalo aplicando las recomendaciones que se estudiaron en esta sección.

```
def v(d):
    vf = (2*9.8*d)**(1/2)
    return vf
```

2.7.8. Más allá de Python

Comparado con otros lenguajes, el formato para la documentación de funciones en Python es relativamente pobre y desestructurado. En Java existe el formato *Javadoc* que es muy estructurado y permite generar automáticamente compendios con la documentación de un programa o una librería. Esquemas similares existen para otros lenguajes como JavaScript (*JSDoc*) y Scala (*Scaladoc*). Aunque no se puedan usar directamente en Python, vale la pena conocer un poco sobre las características de estos formatos (y las limitaciones que tienen) para mejorar la documentación que escribamos de las funciones 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 (el crecimiento de TypeScript es evidencia indiscutible), mientras que Python está empezando a incluir elementos para poder incluir verificaciones de tipos.

[1] Esto no es realmente relevante para este libro, pero lo mencionamos por completitud.

By Mario Sánchez
© Copyright Agosto de 2020.

[Departamento de Ingeniería de Sistemas y Computación](#) - Universidad de los Andes

2.8. Un programa para leer (2)

[Print to PDF](#)
[On this page](#)
[2.8.1. Ejercicios](#)

⚠ 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.

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 raíz 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(cat_1, cat_2)

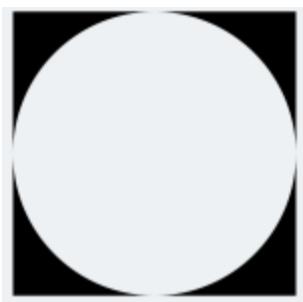
# Mostrarle al usuario el resultado de la operación
print("El perímetro de un triángulo rectángulo que tenga catetos de longitud", cat_1, "y", cat_2,
      "es", perimetro)
```

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?
- ¿Cuál 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?

2.8.1. 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:



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.

By Mario Sánchez
© Copyright Agosto de 2020.

[Departamento de Ingeniería de Sistemas y Computación](#) - Universidad de los Andes

2.9. Lógica vs. Interacción

⚠ 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.

[Print to PDF ▶](#)

ℹ Objetivo de la sección

El objetivo de esta sección es presentar recomendaciones metodológicas para manejar la parte de los programas que se encarga de la interacción con los usuarios y la parte que se encarga de hacer los cálculos y otras operaciones.

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) [1].

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 [2].

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 tremadamente valiosas y que le ahorrarán mucho trabajo en el futuro. Acostúmbrase 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.

2.9.1. 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 los programas 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.

On this page

[2.9.1. Separación de la lógica y la interfaz](#)

[2.9.2. Implementación de módulos separados](#)

[2.9.3. Funciones de la interfaz](#)

[2.9.3.1. Funciones sin parámetros](#)

[2.9.3.2. Funciones sin retorno](#)

[2.9.4. Ejercicios](#)

[2.9.5. Más allá de Python](#)

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 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.

2.9.2. 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 decidíramos ejecutar este módulo usando la siguiente instrucción en la línea de comandos: `python logica_saludar.py`.

The screenshot shows a terminal window with a dark background and light-colored text. At the top, it says 'Default (bash)'. In the center, there is a prompt '\$' followed by the command 'python logica_saludar.py'. Below the command, there is a single green vertical bar representing the cursor or the start of a new line of input.

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 firma 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 'lógica_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 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'.

```
$ python interfaz_saludar.py
¿Cuál es su nombre? Alicia
Hola Alicia!
$
```

2.9.3. 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.

2.9.3.1. 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 firma de la función.

2.9.3.2. 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 firma, 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 [3].

2.9.4. 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`

2.9.5. 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* [4].

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] 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.

[2] 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.

[3] 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.

[4] *Refactoring* es el término utilizado para procesos en los que el código fuente de un programa se reestructura para mejorarlo pero sin cambiar su comportamiento. Por ejemplo, es común que se apliquen operaciones de *refactoring* para hacer que un programa sea mucho más fácil de mantener cambiando el nombre de las funciones para que sean naturales o para que se ajusten a estándares.

By Mario Sánchez
© Copyright Agosto de 2020.

[Departamento de Ingeniería de Sistemas y Computación](#) - Universidad de los Andes

2.10. Errores frecuentes

⚠ 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.

En esta sección presentamos algunos de los errores que encontramos frecuentemente en los programas de muchos estudiantes. Revíselos con cuidado y asegúrese de entenderlos para no cometer esos mismos errores.

2.10.1. Errores con variables y asignaciones

2.10.1.1. Confundir una variable con una cadena

Cuando se empieza a programar, es habitual confundir el nombre de una variable con el valor de una cadena. Recuerde que el nombre de una variable nunca lleva comillas mientras que cualquier cosa entre comillas será un literal.

Considere el siguiente programa. Es más complicado, pero ha sido construido buscando para reflejar la confusión común.

```

1  variable = 'cadena'
2  cadena = 'variable'
3  lenguaje = 'python'
4  print(cadena * 3)
5  print('cadena' * 3)
6  print(python * 5)

```

- En la línea 1, se crea una variable que se llama `variable`, es de tipo cadena (`str`) y tiene el valor '`cadena`'.
- En la línea 2, se crea una variable llamada `cadena`, es de tipo cadena (`str`) y tiene el valor '`variable`'.
- En la línea 3, se crea una variable llamada `lenguaje`, es de tipo cadena (`str`) y tiene el valor '`python`'.
- En la línea 4 se calcula el valor de repetir la variable `cadena` tres veces y se imprime. El programa imprime `variablevariablevariable`.
- En la línea 5 se calcula el valor de repetir la cadena '`cadena`' tres veces y se imprime. El programa imprime `cadenacadenacadena`. Note que en este caso no estamos haciendo referencia a la variable `cadena` sino a la cadena de caracteres '`cadena`'.
- Al intentar ejecutar la línea 6 se produce un error porque no existe ninguna variable con el nombre `python`.

2.10.1.2. Utilizar el valor de una variable a la que no se le ha asignado un valor

Para poder utilizar el valor contenido en una variable, se le debe haber asignado antes un valor. En este contexto, *utilizar* hace referencia a cualquier operación que dependa del valor que tenga una variable. Por ejemplo, en el siguiente programa las dos primeras instrucciones producirían un error mientras que la tercera no tendría ningún problema (suponiendo que antes de ejecutar el programa no exista la variable `v`).

```

1  v += 10
2  n = v * 10
3  v = 55

```

En el caso de las dos primeras instrucciones, el error que se presentaría sería similar al siguiente:

```

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

```

On this page

2.10.1. Errores con variables y asignaciones

2.10.1.1. Confundir una variable con una cadena

2.10.1.2. Utilizar el valor de una variable a la que no se le ha asignado un valor

2.10.1.3. Suponer que una asignación es también una ecuación

2.10.1.4. Intentar hacer una asignación de izquierda a derecha

2.10.2. Errores definiendo funciones

2.10.2.1. No respetar la indentación

2.10.2.2. Usar tabs en lugar de espacios

2.10.2.3. Usar en una función una variable definida en otra función

2.10.2.4. Usar en una función una variable definida por fuera de la función

2.10.2.5. Ponerle a una variable el mismo nombre de una función

2.10.2.6. Redefinir funciones nativas

2.10.3. Errores con el retorno

2.10.3.1. Incluir una instrucción return en una función de tipo None

2.10.3.2. No retornar en una función

2.10.3.3. Escribir instrucciones después del retorno

2.10.4. Otros errores

2.10.4.1. Olvidar el operador de multiplicación

2.10.4.2. Confundir la concatenación de cadenas con la creación de tuplas

2.10.1.3. Suponer que una asignación es también una ecuación

En un ejercicio de cálculo o de física es común que se usen ecuaciones para expresar el valor de una variable. Por ejemplo, la siguiente ecuación expresa el valor de la posición y en términos de la posición inicial y_0 , la velocidad inicial v_{0y} , la aceleración a_y y el tiempo transcurrido:

$$y = y_0 + v_{0y} \cdot t + \frac{1}{2} \cdot a_y \cdot t^2$$

La relación que describe la ecuación entre las variables mencionadas vale en todo momento, antes y después de que nosotros hayamos escrito la ecuación. El hecho de que nosotros la escribamos no cambia absolutamente nada.

Por el contrario, cuando nosotros escribimos una asignación en Python estamos indicando cuál va a ser el valor que tendrá una variable *después de que se ejecute la instrucción*. Considere el siguiente programa:

```
1 x = 27
2 x = y ** 3
```

Si interpretáramos el programa como si fuera un conjunto de ecuaciones, llegaríamos a la conclusión de que tiene que existir alguna variable `y` con el valor `3` para que `x` pueda valer `27` y también pueda valer `y ** 3`.

Si lo interpretamos como lo es, un programa escrito en Python, la situación es muy diferente. Después de ejecutar la línea 1, la variable `x` asume el valor `27`. Luego ejecutamos la línea 2 y le asignamos a `x` un valor que dependerá del valor de `y`. Si `y` tenía el valor `5`, entonces el nuevo valor de `x` será `125` y el valor anterior se habrá perdido.

2.10.1.4. Intentar hacer una asignación de izquierda a derecha

Las asignaciones en Python se hacen de derecha a izquierda: el valor que se encuentre a la derecha del símbolo `=` se almacenará en la variable que se encuentre a la izquierda del símbolo. Si lo que hay a la izquierda no es una variable, se producirá un error. Si lo que hay a la derecha es una expresión, entonces se hará la evaluación de la expresión antes de almacenar el valor en la variable.

2.10.2. Errores definiendo funciones

2.10.2.1. No respetar la indentación

Python es estricto con respecto a la indentación y exige consistencia: si en una función las instrucciones se indentan de forma diferente se producirá un error como el siguiente:

```
IndentationError: unexpected indent
```

2.10.2.2. Usar tabs en lugar de espacios

Una forma común (y fácil) de introducir el error de intentación es usar un carácter de tabulación en lugar de espacios. En este caso, Python nos anunciará algo similar a lo siguiente:

```
TabError: inconsistent use of tabs and spaces in indentation
```

Este error se introduce con facilidad al presionar la tecla tab pero no se detecta fácilmente porque para nuestros ojos una tabulación se ve igual que 4 u 8 espacios. Afortunadamente hoy en día muchos editores de código remplazan automáticamente las tabulaciones por 4 espacios, así que este error se presenta cada vez menos frecuentemente.

2.10.2.3. Usar en una función una variable definida en otra función

Considere el siguiente programa donde se definen dos funciones:

```

1 def fun1(a: int) -> int:
2     doble = a * 2
3     return doble
4
5 def fun2(b: int) -> int:
6     resultado = b + doble
7     return resultado

```

En la línea 6 hay un error porque se está intentando leer el valor de una variable llamada `doble`. Sin embargo, en el *alcance* de esta definición (es decir el espacio de las variables que se podrían leer) no hay ninguna variable con ese nombre. La única variable con ese nombre que vemos en el programa está definida dentro de la función `fun1` pero debemos recordar que las variables que se definen dentro de una función son locales a esa función. Es decir que sólo existen dentro del contexto de esa función.

Si corremos el programa e invocamos a la función `fun2` nos encontraremos con un error como el siguiente:

```
NameError: name 'doble' is not defined
```

2.10.2.4. Usar en una función una variable definida por fuera de la función

A continuación presentamos algo que técnicamente no es un error pero es generalmente considerado una mala práctica de programación porque es fácil que lleve a errores muy difíciles de encontrar y resolver.

Considere el siguiente programa:

```

1 externo = 2
2
3 def fun1(a: int) -> int:
4     valor = a * externo
5     return valor
6
7 print(fun1(5))
8 externo = 3
9 print(fun1(5))

```

Teniendo en cuenta todo lo que hemos dicho hasta ahora, el programa debería fallar cuando se invoque a la función `fun1` porque dentro de la definición de la función no existe una variable o un parámetro llamado `externo`. Sin embargo, el programa anterior funciona porque dado que no se encuentra el valor `externo` dentro del alcance de `fun1`, Python pasa al siguiente alcance (al del módulo que contiene la definición de la función). Como en este alcance, sí hay un valor para la variable `externo`, se usa este valor y no se produce el error al invocar a `fun1`.

Ahora bien, el hecho de que esto funcione no quiere decir que sea una buena idea aprovechar esta posibilidad para no tener que usar parámetros en las funciones. Como dijimos en la sección correspondiente, los parámetros de una función indican qué información es necesaria para ejecutar la función y calcular su resultado. En el programa de ejemplo esto es falso, puesto que el valor `externo` es absolutamente necesario para calcular `fun1`.

También dijimos que, si invocamos una función usando los mismos parámetros, es de esperarse que el resultado sea siempre el mismo. En este caso esto tampoco es cierto: el resultado de `fun1` es diferente en las dos invocaciones debido al cambio en el valor de la variable `externo`.

Aunque no son los únicos argumentos posibles, por ahora deberían ser suficientes para mostrar que escribir funciones que dependan de valores definidos por fuera de la función es una mala idea (salvo casos muy especiales).

2.10.2.5. Ponerle a una variable el mismo nombre de una función

Es un reflejo común usar el nombre de una función para una variable en la que se almacene el resultado de la función, como en este ejemplo:

```

1 def sumatoria(a: int, b: int, c: int) -> int:
2     return a + b + c
3
4 sumatoria = sumatoria(1, 2, 3)
5 sumatoria = sumatoria(4, 5, 6)

```

En este caso, el programa va a funcionar hasta la línea 4 y va a fallar en la línea 5 con un error como el siguiente:

```
TypeError: 'int' object is not callable
```

Sin embargo, el error se introdujo realmente en la línea anterior. Analicemos con cuidado lo que ocurre en esta línea:

1. Se hace una invocación a la función `sumatoria` usando los parámetros 1, 2 y 3.
2. El resultado de la invocación (el entero 6) se almacena en la variable `sumatoria` - que no existía antes de este momento.

En la siguiente línea se intenta hacer una invocación a `sumatoria`, pero Python lo último que vio con ese nombre no era una función sino una variable de tipo `int`, por lo cual genera el error diciendo que no es posible hacer una invocación sobre un entero.

2.10.2.6. Redefinir funciones nativas

A menos que haya alguna necesidad muy especial, nunca se deberían redefinir las funciones nativas de Python, tales como float, int, str, min, max y abs, entre otras. Sin embargo, en Python esto es realmente muy fácil de hacer y por lo tanto ocurre con frecuencia cuando se está empezando a programar.

Considere el siguiente ejemplo:

```

1 def menor(a: int, b: int, c: int) -> int:
2     return min(min(a, b), c)
3
4 min = menor(5,7,3)
5 verificacion = min(5, 7, 3)
6 print(min, verificacion)

```

En esta función se define la función `menor` que busca el menor de 3 números aplicando la función `min` entre los dos primeros y luego la función `min` entre el tercero y el menor entre los dos primeros números. En la línea 4 se invoca a la función `menor` usando los valores 5, 7 y 3 y se almacena el resultado en la variable `min`. En la línea 5, se usa la función `min` con 3 parámetros para ver si el resultado es el mismo que con nuestra nueva función `menor`. Al ejecutar la línea 5 se produce el error y no se puede hacer la invocación:

```
TypeError: 'int' object is not callable
```

Esto ocurre porque en la línea 4 creamos una variable llamada `min` de tipo entero y ahora Python no puede encontrar la función nativa `min`. El error en este caso es sencillo de corregir, pero también es fácil de introducir más adelante.

2.10.3. Errores con el retorno

2.10.3.1. Incluir una instrucción `return` en una función de tipo `None`

Si en la signatira de una función se declara que la función es de tipo `None`, no debería haber una instrucción `return` dentro de la función.

2.10.3.2. No retornar en una función

Fuera de las funciones que declaran en la signatura que son de tipo `None`, todas las funciones deberían incluir una instrucción `return`. Es recomendable, aunque no obligatorio, que sólo haya una instrucción `return` al final de la función.

2.10.3.3. Escribir instrucciones después del retorno

Las instrucciones que se escriban después de la instrucción `return` no se ejecutarán en un programa escribir instrucciones después del retorno

2.10.4. Otros errores

2.10.4.1. Olvidar el operador de multiplicación

Considere el siguiente programa:

```
1 a = m(m+1)/2
2 b = n*(n+1)/2
```

Probablemente el que escribió la primera línea quería realizar el mismo cálculo que en la línea 2 pero se le olvidó incluir el operador de multiplicación. Eso quiere decir que en la primera línea Python va a intentar invocar a la función `m` pasándole como parámetro el valor de `m+1`. Muy posiblemente esto no va a funcionar.

2.10.4.2. Confundir la concatenación de cadenas con la creación de tuplas

Más adelante en este libro estudiaremos un tipo de datos llamado tupla. Por ahora es suficiente decir que las tuplas son el tipo que uno usaría para representar cosas como coordenadas en un plano cartesiano (que tienen varias partes). En el siguiente código, en la primera línea se define una coordenada usando una tupla: note el uso de los paréntesis y de la coma para separar los valores 3 y 5.

```
1 coordenada_2D = (3, 5)
2 nombre_1 = 'Alberto' + ',' + 'García' # nombre_1 es un str
3 nombre_2 = 'Alberto', 'García'       # nombre_2 es una tupla
```

La segunda línea del programa crea una variable de tipo `str` con el valor `"Alberto,García"`. Esto ocurre porque utilizamos el operador `+` entre cadenas de caracteres (concatenación) y concatenamos una cadena que tenía una coma `(",")` entre el nombre y el apellido.

En la tercera línea del programa no utilizamos el operador de concatenación, pero pusimos una coma entre el nombre y el apellido. En muchos lenguajes de programación se marcaría como un error de sintaxis, pero en Python esto corresponde a la creación de una tupla: la diferencia con la línea 1 es que en este caso no se utilizaron los paréntesis, que son opcionales cuando se crea una tupla, y que la nueva tupla tiene dos cadenas por dentro en lugar de dos números.

No se preocupe si no entiende lo que es una tupla en este momento. El punto importante es que no confunda una tupla con una cadena donde las partes estén separadas con comas.

2.11. Para no olvidar

- Cada línea en programa debería tener una instrucción. Es más fácil de leer, entender, mantener y corregir un programa que tenga varias instrucciones sencillas, en lugar de uno donde cada línea intenta hacer varias cosas a la vez.
- Las instrucciones de un mismo bloque se ejecutan una por una. Un bloque hace referencia a las instrucciones que están contenidas dentro del mismo módulo o de la misma función y que tienen el mismo nivel de indentación. Las instrucciones de un bloque se ejecutan desde la primera línea hasta la última, una por una. No son ecuaciones que tengan que ser ciertas todas a la vez.
- Definir una función no es invocarla. A pesar de que se haya definido en la parte de arriba de un módulo, una función no necesariamente se va a ejecutar primero.
- En la parte izquierda de una asignación siempre va una variable. No puede haber literales, ni expresiones, ni invocaciones de funciones en la parte izquierda de una asignación.
- En una asignación, lo primero que se hace es evaluar el valor que está en la parte derecha. Sólo cuando ya se tiene un valor se asigna en la variable de la izquierda.
- Es mejor usar nombres de variables explícitos y que sugieran su tipo. Por ejemplo, para guardar el teléfono de una persona es mejor utilizar una variable que se llame `telefono` a una variable que se llame `numero`. Esta última sería problemática si usáramos una cadena de caracteres, lo cual es muy usual dado que muchos teléfonos

tienen otros caracteres además de los dígitos (como en [+57 3394949 ext 2860](#)).

- Las funciones se invocan con su nombre y los valores para los parámetros dentro de paréntesis. Si una función no recibe parámetros, de todas formas es necesario incluir los paréntesis.
- Los valores de los parámetros no tienen que llamarse igual que las variables que se usan al invocar la función. Por ejemplo, si tenemos la función con signatura `{python} def f(a: int, b: float)-> float`, no es necesario que tengamos variables llamadas `a` y `b` para hacer la invocación. En el siguiente programa todas las invocaciones serían correctas:

```
1 a = 1
2 b = 4.5
3 c = 7
4 r1 = f(a, b)
5 r2 = f(c, b)
6 r3 = f(c, 9.9)
```

- Es una buena práctica usar comentarios. Si tiene dudas sobre si debería incluir un comentario en un cierto punto, mejor inclúyalo. Sea breve pero preciso en sus comentarios.

By Mario Sánchez

© Copyright Agosto de 2020.

[Departamento de Ingeniería de Sistemas y Computación](#) - Universidad de los Andes

[Print to PDF ►](#)

2.10. Errores frecuentes

⚠ 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.

En esta sección presentamos algunos de los errores que encontramos frecuentemente en los programas de muchos estudiantes. Revíselos con cuidado y asegúrese de entenderlos para no cometer esos mismos errores.

2.10.1. Errores con variables y asignaciones

2.10.1.1. Confundir una variable con una cadena

Cuando se empieza a programar, es habitual confundir el nombre de una variable con el valor de una cadena. Recuerde que el nombre de una variable nunca lleva comillas mientras que cualquier cosa entre comillas será un literal.

Considere el siguiente programa. Es más complicado, pero ha sido construido buscando para reflejar la confusión común.

```

1  variable = 'cadena'
2  cadena = 'variable'
3  lenguaje = 'python'
4  print(cadena * 3)
5  print('cadena' * 3)
6  print(python * 5)

```

- En la línea 1, se crea una variable que se llama `variable`, es de tipo cadena (`str`) y tiene el valor '`'cadena'`'.
- En la línea 2, se crea una variable llamada `cadena`, es de tipo cadena (`str`) y tiene el valor '`'variable'`'.
- En la línea 3, se crea una variable llamada `lenguaje`, es de tipo cadena (`str`) y tiene el valor '`'python'`'.
- En la línea 4 se calcula el valor de repetir la variable `cadena` tres veces y se imprime. El programa imprime `variablevariablevariable`.
- En la línea 5 se calcula el valor de repetir la cadena '`'cadena'`' tres veces y se imprime. El programa imprime `cadenacadenacadena`. Note que en este caso no estamos haciendo referencia a la variable `cadena` sino a la cadena de caracteres '`'cadena'`'.
- Al intentar ejecutar la línea 6 se produce un error porque no existe ninguna variable con el nombre `python`.

2.10.1.2. Utilizar el valor de una variable a la que no se le ha asignado un valor

Para poder utilizar el valor contenido en una variable, se le debe haber asignado antes un valor. En este contexto, *utilizar* hace referencia a cualquier operación que dependa del valor que tenga una variable. Por ejemplo, en el siguiente programa las dos primeras instrucciones producirían un error mientras que la tercera no tendría ningún problema (suponiendo que antes de ejecutar el programa no exista la variable `v`).

```

1  v += 10
2  n = v * 10
3  v = 55

```

En el caso de las dos primeras instrucciones, el error que se presentaría sería similar al siguiente:

```

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

```

On this page

2.10.1. Errores con variables y asignaciones

2.10.1.1. Confundir una variable con una cadena

2.10.1.2. Utilizar el valor de una variable a la que no se le ha asignado un valor

2.10.1.3. Suponer que una asignación es también una ecuación

2.10.1.4. Intentar hacer una asignación de izquierda a derecha

2.10.2. Errores definiendo funciones

2.10.2.1. No respetar la indentación

2.10.2.2. Usar tabs en lugar de espacios

2.10.2.3. Usar en una función una variable definida en otra función

2.10.2.4. Usar en una función una variable definida por fuera de la función

2.10.2.5. Ponerle a una variable el mismo nombre de una función

2.10.2.6. Redefinir funciones nativas

2.10.3. Errores con el retorno

2.10.3.1. Incluir una instrucción `return` en una función de tipo `None`

2.10.3.2. No retornar en una función

2.10.3.3. Escribir instrucciones después del retorno

2.10.4. Otros errores

2.10.4.1. Olvidar el operador de multiplicación

2.10.4.2. Confundir la concatenación de cadenas con la creación de tuplas

2.10.1.3. Suponer que una asignación es también una ecuación

En un ejercicio de cálculo o de física es común que se usen ecuaciones para expresar el valor de una variable. Por ejemplo, la siguiente ecuación expresa el valor de la posición y en términos de la posición inicial y_0 , la velocidad inicial v_{0y} , la aceleración a_y y el tiempo transcurrido:

$$y = y_0 + v_{0y} \cdot t + \frac{1}{2} \cdot a_y \cdot t^2$$

La relación que describe la ecuación entre las variables mencionadas vale en todo momento, antes y después de que nosotros hayamos escrito la ecuación. El hecho de que nosotros la escribamos no cambia absolutamente nada.

Por el contrario, cuando nosotros escribimos una asignación en Python estamos indicando cuál va a ser el valor que tendrá una variable *después de que se ejecute la instrucción*. Considere el siguiente programa:

```
1 x = 27
2 x = y ** 3
```

Si interpretáramos el programa como si fuera un conjunto de ecuaciones, llegaríamos a la conclusión de que tiene que existir alguna variable `y` con el valor `3` para que `x` pueda valer `27` y también pueda valer `y ** 3`.

Si lo interpretamos como lo es, un programa escrito en Python, la situación es muy diferente. Después de ejecutar la línea 1, la variable `x` asume el valor `27`. Luego ejecutamos la línea 2 y le asignamos a `x` un valor que dependerá del valor de `y`. Si `y` tenía el valor `5`, entonces el nuevo valor de `x` será `125` y el valor anterior se habrá perdido.

2.10.1.4. Intentar hacer una asignación de izquierda a derecha

Las asignaciones en Python se hacen de derecha a izquierda: el valor que se encuentre a la derecha del símbolo `=` se almacenará en la variable que se encuentre a la izquierda del símbolo. Si lo que hay a la izquierda no es una variable, se producirá un error. Si lo que hay a la derecha es una expresión, entonces se hará la evaluación de la expresión antes de almacenar el valor en la variable.

2.10.2. Errores definiendo funciones

2.10.2.1. No respetar la indentación

Python es estricto con respecto a la indentación y exige consistencia: si en una función las instrucciones se indentan de forma diferente se producirá un error como el siguiente:

```
IndentationError: unexpected indent
```

2.10.2.2. Usar tabs en lugar de espacios

Una forma común (y fácil) de introducir el error de intentación es usar un carácter de tabulación en lugar de espacios. En este caso, Python nos anunciará algo similar a lo siguiente:

```
TabError: inconsistent use of tabs and spaces in indentation
```

Este error se introduce con facilidad al presionar la tecla tab pero no se detecta fácilmente porque para nuestros ojos una tabulación se ve igual que 4 u 8 espacios. Afortunadamente hoy en día muchos editores de código remplazan automáticamente las tabulaciones por 4 espacios, así que este error se presenta cada vez menos frecuentemente.

2.10.2.3. Usar en una función una variable definida en otra función

Considere el siguiente programa donde se definen dos funciones:

```

1 def fun1(a: int) -> int:
2     doble = a * 2
3     return doble
4
5 def fun2(b: int) -> int:
6     resultado = b + doble
7     return resultado

```

En la línea 6 hay un error porque se está intentando leer el valor de una variable llamada `doble`. Sin embargo, en el *alcance* de esta definición (es decir el espacio de las variables que se podrían leer) no hay ninguna variable con ese nombre. La única variable con ese nombre que vemos en el programa está definida dentro de la función `fun1` pero debemos recordar que las variables que se definen dentro de una función son locales a esa función. Es decir que sólo existen dentro del contexto de esa función.

Si corremos el programa e invocamos a la función `fun2` nos encontraremos con un error como el siguiente:

```
NameError: name 'doble' is not defined
```

2.10.2.4. Usar en una función una variable definida por fuera de la función

A continuación presentamos algo que técnicamente no es un error pero es generalmente considerado una mala práctica de programación porque es fácil que lleve a errores muy difíciles de encontrar y resolver.

Considere el siguiente programa:

```

1 externo = 2
2
3 def fun1(a: int) -> int:
4     valor = a * externo
5     return valor
6
7 print(fun1(5))
8 externo = 3
9 print(fun1(5))

```

Teniendo en cuenta todo lo que hemos dicho hasta ahora, el programa debería fallar cuando se invoque a la función `fun1` porque dentro de la definición de la función no existe una variable o un parámetro llamado `externo`. Sin embargo, el programa anterior funciona porque dado que no se encuentra el valor `externo` dentro del alcance de `fun1`, Python pasa al siguiente alcance (al del módulo que contiene la definición de la función). Como en este alcance, sí hay un valor para la variable `externo`, se usa este valor y no se produce el error al invocar a `fun1`.

Ahora bien, el hecho de que esto funcione no quiere decir que sea una buena idea aprovechar esta posibilidad para no tener que usar parámetros en las funciones. Como dijimos en la sección correspondiente, los parámetros de una función indican qué información es necesaria para ejecutar la función y calcular su resultado. En el programa de ejemplo esto es falso, puesto que el valor `externo` es absolutamente necesario para calcular `fun1`.

También dijimos que, si invocamos una función usando los mismos parámetros, es de esperarse que el resultado sea siempre el mismo. En este caso esto tampoco es cierto: el resultado de `fun1` es diferente en las dos invocaciones debido al cambio en el valor de la variable `externo`.

Aunque no son los únicos argumentos posibles, por ahora deberían ser suficientes para mostrar que escribir funciones que dependan de valores definidos por fuera de la función es una mala idea (salvo casos muy especiales).

2.10.2.5. Ponerle a una variable el mismo nombre de una función

Es un reflejo común usar el nombre de una función para una variable en la que se almacene el resultado de la función, como en este ejemplo:

```

1 def sumatoria(a: int, b: int, c: int) -> int:
2     return a + b + c
3
4 sumatoria = sumatoria(1, 2, 3)
5 sumatoria = sumatoria(4, 5, 6)

```

En este caso, el programa va a funcionar hasta la línea 4 y va a fallar en la línea 5 con un error como el siguiente:

```
TypeError: 'int' object is not callable
```

Sin embargo, el error se introdujo realmente en la línea anterior. Analicemos con cuidado lo que ocurre en esta línea:

1. Se hace una invocación a la función `sumatoria` usando los parámetros 1, 2 y 3.
2. El resultado de la invocación (el entero 6) se almacena en la variable `sumatoria` - que no existía antes de este momento.

En la siguiente línea se intenta hacer una invocación a `sumatoria`, pero Python lo último que vio con ese nombre no era una función sino una variable de tipo `int`, por lo cual genera el error diciendo que no es posible hacer una invocación sobre un entero.

2.10.2.6. Redefinir funciones nativas

A menos que haya alguna necesidad muy especial, nunca se deberían redefinir las funciones nativas de Python, tales como float, int, str, min, max y abs, entre otras. Sin embargo, en Python esto es realmente muy fácil de hacer y por lo tanto ocurre con frecuencia cuando se está empezando a programar.

Considere el siguiente ejemplo:

```

1 def menor(a: int, b: int, c: int) -> int:
2     return min(min(a, b), c)
3
4 min = menor(5,7,3)
5 verificacion = min(5, 7, 3)
6 print(min, verificacion)

```

En esta función se define la función `menor` que busca el menor de 3 números aplicando la función `min` entre los dos primeros y luego la función `min` entre el tercero y el menor entre los dos primeros números. En la línea 4 se invoca a la función `menor` usando los valores 5, 7 y 3 y se almacena el resultado en la variable `min`. En la línea 5, se usa la función `min` con 3 parámetros para ver si el resultado es el mismo que con nuestra nueva función `menor`. Al ejecutar la línea 5 se produce el error y no se puede hacer la invocación:

```
TypeError: 'int' object is not callable
```

Esto ocurre porque en la línea 4 creamos una variable llamada `min` de tipo entero y ahora Python no puede encontrar la función nativa `min`. El error en este caso es sencillo de corregir, pero también es fácil de introducir más adelante.

2.10.3. Errores con el retorno

2.10.3.1. Incluir una instrucción `return` en una función de tipo `None`

Si en la signatira de una función se declara que la función es de tipo `None`, no debería haber una instrucción `return` dentro de la función.

2.10.3.2. No retornar en una función

Fuera de las funciones que declaran en la signatura que son de tipo `None`, todas las funciones deberían incluir una instrucción `return`. Es recomendable, aunque no obligatorio, que sólo haya una instrucción `return` al final de la función.

2.10.3.3. Escribir instrucciones después del retorno

Las instrucciones que se escriban después de la instrucción `return` no se ejecutarán en un programa escribir instrucciones después del retorno

2.10.4. Otros errores

2.10.4.1. Olvidar el operador de multiplicación

Considere el siguiente programa:

```
1 a = m(m+1)/2
2 b = n*(n+1)/2
```

Probablemente el que escribió la primera línea quería realizar el mismo cálculo que en la línea 2 pero se le olvidó incluir el operador de multiplicación. Eso quiere decir que en la primera línea Python va a intentar invocar a la función `m` pasándole como parámetro el valor de `m+1`. Muy posiblemente esto no va a funcionar.

2.10.4.2. Confundir la concatenación de cadenas con la creación de tuplas

Más adelante en este libro estudiaremos un tipo de datos llamado tupla. Por ahora es suficiente decir que las tuplas son el tipo que uno usaría para representar cosas como coordenadas en un plano cartesiano (que tienen varias partes). En el siguiente código, en la primera línea se define una coordenada usando una tupla: note el uso de los paréntesis y de la coma para separar los valores 3 y 5.

```
1 coordenada_2D = (3, 5)
2 nombre_1 = 'Alberto' + ',' + 'García' # nombre_1 es un str
3 nombre_2 = 'Alberto', 'García'       # nombre_2 es una tupla
```

La segunda línea del programa crea una variable de tipo `str` con el valor `"Alberto,García"`. Esto ocurre porque utilizamos el operador `+` entre cadenas de caracteres (concatenación) y concatenamos una cadena que tenía una coma `(",")` entre el nombre y el apellido.

En la tercera línea del programa no utilizamos el operador de concatenación, pero pusimos una coma entre el nombre y el apellido. En muchos lenguajes de programación se marcaría como un error de sintaxis, pero en Python esto corresponde a la creación de una tupla: la diferencia con la línea 1 es que en este caso no se utilizaron los paréntesis, que son opcionales cuando se crea una tupla, y que la nueva tupla tiene dos cadenas por dentro en lugar de dos números.

No se preocupe si no entiende lo que es una tupla en este momento. El punto importante es que no confunda una tupla con una cadena donde las partes estén separadas con comas.

2.11. Para no olvidar

- Cada línea en programa debería tener una instrucción. Es más fácil de leer, entender, mantener y corregir un programa que tenga varias instrucciones sencillas, en lugar de uno donde cada línea intenta hacer varias cosas a la vez.
- Las instrucciones de un mismo bloque se ejecutan una por una. Un bloque hace referencia a las instrucciones que están contenidas dentro del mismo módulo o de la misma función y que tienen el mismo nivel de indentación. Las instrucciones de un bloque se ejecutan desde la primera línea hasta la última, una por una. No son ecuaciones que tengan que ser ciertas todas a la vez.
- Definir una función no es invocarla. A pesar de que se haya definido en la parte de arriba de un módulo, una función no necesariamente se va a ejecutar primero.
- En la parte izquierda de una asignación siempre va una variable. No puede haber literales, ni expresiones, ni invocaciones de funciones en la parte izquierda de una asignación.
- En una asignación, lo primero que se hace es evaluar el valor que está en la parte derecha. Sólo cuando ya se tiene un valor se asigna en la variable de la izquierda.
- Es mejor usar nombres de variables explícitos y que sugieran su tipo. Por ejemplo, para guardar el teléfono de una persona es mejor utilizar una variable que se llame `telefono` a una variable que se llame `numero`. Esta última sería problemática si usáramos una cadena de caracteres, lo cual es muy usual dado que muchos teléfonos

tienen otros caracteres además de los dígitos (como en [+57 3394949 ext 2860](#)).

- Las funciones se invocan con su nombre y los valores para los parámetros dentro de paréntesis. Si una función no recibe parámetros, de todas formas es necesario incluir los paréntesis.
- Los valores de los parámetros no tienen que llamarse igual que las variables que se usan al invocar la función. Por ejemplo, si tenemos la función con signatura `{python} def f(a: int, b: float)-> float`, no es necesario que tengamos variables llamadas `a` y `b` para hacer la invocación. En el siguiente programa todas las invocaciones serían correctas:

```
1 a = 1
2 b = 4.5
3 c = 7
4 r1 = f(a, b)
5 r2 = f(c, b)
6 r3 = f(c, 9.9)
```

- Es una buena práctica usar comentarios. Si tiene dudas sobre si debería incluir un comentario en un cierto punto, mejor inclúyalo. Sea breve pero preciso en sus comentarios.

By Mario Sánchez

© Copyright Agosto de 2020.

[Departamento de Ingeniería de Sistemas y Computación](#) - Universidad de los Andes

[Print to PDF ►](#)

2.12. Ejercicios adicionales

[Print to PDF ▶](#)

≡ On this page

- [2.12.1. Conceptos básicos](#)
- [2.12.2. Lectura de programas](#)
- [2.12.3. Búsqueda de errores](#)
- [2.12.3.1. Tiempo de descarga](#)
- [2.12.3.2. IVA y propina](#)
- [2.12.3.3. Saludo prolongado](#)
- [2.12.4. Solución de problemas](#)

2.12.1. Conceptos básicos

1. Defina y dé un ejemplo para cada uno de los siguientes conceptos que se estudiaron en este capítulo:

- Tipo de datos
- Variable
- Expresión
- Operador
- Función
- Parámetro

2. ¿Qué tipo de dato utilizaría para representar cada uno de los siguientes elementos?

- El nombre de una persona
- El número de identificación de una persona
- La edad de una persona
- El cilindraje del motor de un carro
- La capacidad en litros del baúl de un carro
- El año de fabricación de un carro
- La marca de un carro

2.12.2. Lectura de programas

¿Qué imprimen los siguientes programas? Intente predecir qué van a imprimir los siguientes programas y después copie el código al intérprete de Python y verifique sus respuestas.

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

```
v1 = 5
v2 = v1 ** 2
v2 = v1 + v2
print("v2:", v2)
```

```
v1 = '5'
v2 = v1 * 3
print("v2:", v2)
```

```
v1 = '5'
v2 = v1 * 3
v3 = v1 + v2
print("v3:", v3)
```

```
v1 = '5'
v2 = v1 * 3
v3 = v1 + v2 + 9
print("v3:", v3)
```

```
v1 = 124
v2 = v1 // 3
v3 = v1 % 3
print("124/3:", v2, v3)
```

2.12.3. Búsqueda de errores

Revise con atención los siguientes enunciados y el programa que intenta resolverlos. Explique cuál es el problema del programa y corríjalo.

Ayuda: Si no encuentra los problemas leyendo el código, copie los programas a su IDE y ejecútelo para intentar diagnosticar los problemas. Para esto tendrá que identificar valores de los parámetros que le sean útiles y posiblemente hacer los cálculos *a mano* para ver qué podría estar mal.

2.12.3.1. Tiempo de descarga

Enunciado: Escriba una función que reciba la velocidad del Internet de un usuario (en Mbps, es decir, Megabits por segundo), y el tamaño de un archivo a descargar (en GB, es decir, Gigabytes), y retorne el tiempo en minutos estimado (redondee al entero más cercano) para realizar la descarga de dicho archivo sobre esa red. Para esto, tenga en cuenta que el tiempo lo puede calcular como $t = \text{tamaño_archivo} / \text{velocidad_descarga}$. El tamaño y velocidad deben estar en unidades homogéneas (por ejemplo, MB y MB/s, o GB y GB/s) para que se puedan operar). *Nota:* Recuerde que un MB (Megabyte) equivale a 8 Mb (Megabits), y que un GB equivale a 1000 MB.

```
def calcular_tiempo_descarga(velocidad: int, tamano_archivo: int)->int:
    """ Tiempo de descarga
    Parámetros:
        velocidad (int): Velocidad de descarga de la red, en Mbps
        tamano_archivo (int): Tamaño del archivo a descargar, en GB
    Retorno:
        int: Tiempo estimado en minutos que toma la descarga del archivo
    """
    gb_megas = tamano_archivo * 1000
    v = velocidad * 8
    tiempo = gb_megas / v
    return round(tiempo)
```

2.12.3.2. IVA y propina

Enunciado: Escriba una función que reciba el costo en pesos de una cuenta de restaurante, y luego calcule el impuesto IVA asociado y la propina para el mesero. La tasa del IVA corresponde al 19%, y la propina en el restaurante es del 10% del valor de la factura (sin impuestos). Debe retornar una cadena que muestre el IVA, propina y total de la siguiente manera: "X,Y,Z", donde X es el IVA, Y la propina y Z el total. No olvide aproximar los números al entero más cercano.

```
def calcular_iva_propina_total_factura (costo_factura: int) -> str:
    """ IVA y propina
    Parámetros:
        costo_factura (int): Costo de la factura del restaurante, sin impuestos ni propina
    Retorno:
        str: Cadena con el iva, propina y total de la factura, separados por coma
    """
    IVA = round (costo_factura*19/100)
    propina = round (costo_factura*10/100)
    total = round (costo_factura + propina + IVA)
    return str(IVA) + "," + str(propina) + "," + str(total)
```

2.12.3.3. Saludo prolongado

Enunciado: Escriba una función que reciba un nombre y un entero, y retorne la cadena 'Hola' seguida del nombre recibido por parámetro, pero con la letra 'o' repetida tantas veces como indique el entero recibido, así como la letra 'a' la mitad de las veces que la 'o' (si la mitad no es exacta, se debe tomar la parte entera). Por ejemplo, si se reciben como parámetros 'Egan' y 5, la cadena a retornar será 'Hooooolaa Egan'.

```
def saludar_repetidas_veces(nombre: str, veces: int)->str:
    """ Saludo prolongado
    Parámetros:
        nombre (str): Nombre a incluir en el saludo
        veces (int): Cantidad de veces a repetir las letras
    Retorno:
        str: Cadena con el saludo con letras repetidas
    """
    o_varias_veces = "o" * veces
    a_varias_veces = "a" * (veces//2)
    return "h" + o_varias_veces + "l" + a_varias_veces + "+" + nombre
```

2.12.4. Solución de problemas

1. Enunciado: El volumen de un cilindro se puede calcular multiplicando el área de su base circular por su altura.
Cree una función que reciba el radio de la base y la altura del cilindro, y calcule su volumen. Retorne el resultado redondeado a un solo decimal.
2. Enunciado: Cree una función que pueda calcular el índice de masa corporal (BMI) de una persona a partir de su peso en libras y su altura en pulgadas. La fórmula para calcular el BMI es la siguiente:
 $BMI = peso / (altura^2)$, pero para que la fórmula funcione *peso* debe estar en kilogramos y *altura* en metros. Recuerde que 1 libra corresponde a 0.454 kg, y que 1 pulgada corresponde a 0.0254 metros. El valor de retorno debe estar redondeado a dos decimales.

By Mario Sánchez

© Copyright Agosto de 2020.

[Departamento de Ingeniería de Sistemas y Computación](#) - Universidad de los Andes

3.1. Un programa para leer

[Print to PDF ▶](#)[On this page](#)[3.1.1. Lógica de la aplicación](#)[3.1.2. Interfaz de la aplicación](#)

⚠ 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.

Como dijimos en el primer capítulo, una habilidad importante que se debe desarrollar es la capacidad para leer código escrito por alguien más. En esta sección presentamos un nuevo ejemplo para que se ejercente y para que también observe varias de las características de Python que se van a estudiar en este nivel.

Al igual que en el nivel 1, el programa no tiene comentarios para que el ejercicio de lectura sea más exigente pero también más útil. Un programa tan largo como este normalmente tendría una buena cantidad de comentarios.

3.1.1. Lógica de la aplicación

El siguiente es el contenido completo del módulo [logica_juego1](#).

```

import random

def mano_nueva()->dict:
    mano = {'carta1': carta_nueva(), 'carta2': carta_nueva(), 'tam': 2}
    return mano

def carta_nueva()->dict:
    palo = nombre_palo(random.randint(1,4))
    valor = random.randint(1,13)
    nombre_carta = "{} de {}".format(nombre_valor(valor), palo)
    carta = {"palo": palo, "valor": valor, "nombre": nombre_carta}
    return carta

def agregar_carta(mano: dict)->dict:
    tam_actual = mano['tam']
    nueva_carta = carta_nueva()
    nueva_llave = 'carta{}'.format(tam_actual + 1)
    mano[nueva_llave] = nueva_carta
    mano['tam'] = tam_actual + 1
    return nueva_carta

def nombre_palo(num_palo: int)->str:
    nombre = "Picas"
    if num_palo == 2:
        nombre = "Corazones"
    elif num_palo == 3:
        nombre = "Tréboles"
    elif num_palo == 4:
        nombre = "Diamantes"
    return nombre

def nombre_valor(valor: int)->str:
    nombre = str(valor)
    if valor == 1:
        nombre = "AS"
    elif valor == 11:
        nombre = "J"
    elif valor == 12:
        nombre = "Q"
    elif valor == 13:
        nombre = "K"
    return nombre

def contar_puntos_mano(mano: dict)->int:
    puntos = contar_puntos_carta(mano, 1)
    puntos += contar_puntos_carta(mano, 2)
    puntos += contar_puntos_carta(mano, 3)
    puntos += contar_puntos_carta(mano, 4)
    puntos += contar_puntos_carta(mano, 5)
    return puntos

def contar_puntos_carta(mano: dict, numero_carta: int)->int:
    puntos = 0
    llave = "carta{}".format(numero_carta)
    if llave in mano:
        carta = mano[llave]
        valor = carta["valor"]
        if valor == 1:
            puntos = 11
        elif valor > 10:
            puntos = 10
        else:
            puntos = valor
    return puntos

def casa_debe_continuar(mano_casa: dict)->bool:
    puntos_casa = contar_puntos_mano(mano_casa)
    return puntos_casa < 16 or puntos_casa > 21

```

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 cree que es el objetivo del programa?
- ¿Cuál es el objetivo de cada bloque?
- ¿Qué elementos de la sintaxis utilizada no conoce todavía?

3.1.2. Interfaz de la aplicación

El siguiente es el contenido completo del módulo [interfaz_juego1](#).

```

import logica_juego1 as juego

def mostrar_menu()->int:
    print("\n¿Cómo quiere continuar?")
    print(" 1. Quiero otra carta")
    print(" 2. No quiero más cartas")
    respuesta = input("Seleccione una opción: ")
    if respuesta.isnumeric():
        opcion = int(respuesta)
        if opcion != 1 and opcion != 2:
            print("Sólo podía seleccionar 1 o 2. Suponemos que no quiere más cartas")
            opcion = 2
    else:
        print("Sólo podía seleccionar 1 o 2. Suponemos que no quiere más cartas")
        opcion = 2
    return opcion

def mostrar_primera_carta(mano: dict)->None:
    plantilla = "La mano tiene {} cartas y está mostrando un {}"
    print(plantilla.format(mano['tam'], mano['carta1']['nombre']))

def mostrar_mano_abierta(mano: dict)->None:
    num_cartas = mano['tam']
    puntos_mano = juego.contar_puntos_mano(mano)
    plantilla = "La mano tiene {} cartas y vale {} puntos"
    print(plantilla.format(num_cartas, puntos_mano))
    print(" La primera carta es: ", mano['carta1']['nombre'])
    print(" La segunda carta es: ", mano['carta2']['nombre'])

    if num_cartas >= 3:
        print(" La tercera carta es: ", mano['carta3']['nombre'])

    if num_cartas >= 4:
        print(" La cuarta carta es: ", mano['carta4']['nombre'])

    if num_cartas >= 5:
        print(" La quinta carta es: ", mano['carta5']['nombre'])

def mostrar_juego_actual(mano_jugador: dict, mano_casa: dict, abrir_casa: bool = False):
    print("MANO JUGADOR")
    mostrar_mano_abierta(mano_jugador)
    print("MANO CASA")
    if abrir_casa:
        mostrar_mano_abierta(mano_casa)
    else:
        mostrar_primera_carta(mano_casa)

def turno_jugador(mano_jugador: dict, mano_casa: dict)->bool:
    continuar_juego = True
    opcion = mostrar_menu()
    if opcion == 2:
        continuar_juego = False
    else:
        nueva = juego.agregar_carta(mano_jugador)
        print("Tu nueva carta es: ", nueva['nombre'])
        mostrar_juego_actual(mano_jugador, mano_casa)
        puntos_mano = juego.contar_puntos_mano(mano_jugador)
        if puntos_mano > 21:
            print("\nTienes más de 21 puntos: ¡Perdiste!")
            continuar_juego = False
        elif puntos_mano == 21:
            print("\nTienes exactamente 21 puntos: esperemos que la casa no tenga lo mismo.\n")
            continuar_juego = False

    return continuar_juego

def turno_casa(puntos_jugador: int, mano_casa: dict)->bool:
    continuar_juego = True
    puntos_casa = juego.contar_puntos_mano(mano_casa)
    if puntos_casa > puntos_jugador:
        print("La casa gana: {1} de la casa vs {0} del jugador".format(puntos_jugador, puntos_casa))
        continuar_juego = False
    else:
        if juego.casa_debe_continuar(mano_casa):
            nueva = juego.agregar_carta(mano_casa)
            print("La casa saca otra carta ... {}".format(nueva['nombre']))
            mostrar_mano_abierta(mano_casa)

            puntos_casa = juego.contar_puntos_mano(mano_casa)
            if puntos_casa > 21:
                print("\nLa casa tiene más de 21: ¡el jugador gana!\n")
                continuar_juego = False

```

```

    elif puntos_casa == 21:
        print("\nLa casa tiene 21: ¡la casa gana!\n")
        continuar_juego = False

    return continuar_juego

def iniciar_aplicacion() -> None:
    mano_jugador = juego.mano_nueva()
    mano_casa = juego.mano_nueva()
    mostrar_juego_actual(mano_jugador, mano_casa)

    continuar_juego = True

    # tercera carta del jugador
    if continuar_juego:
        continuar_juego = turno_jugador(mano_jugador, mano_casa)

    # cuarta carta del jugador
    if continuar_juego:
        continuar_juego = turno_jugador(mano_jugador, mano_casa)

    # quinta carta del jugador
    if continuar_juego:
        continuar_juego = turno_jugador(mano_jugador, mano_casa)

    puntos_jugador = juego.contar_puntos_mano(mano_jugador)
    if puntos_jugador <= 21:
        mostrar_juego_actual(mano_jugador, mano_casa, True)
        print("\nEs el turno de la casa de jugar\n")
        continuar_juego = True

    # tercera carta de La casa
    if continuar_juego:
        continuar_juego = turno_casa(puntos_jugador, mano_casa)

    # cuarta carta de La casa
    if continuar_juego:
        continuar_juego = turno_casa(puntos_jugador, mano_casa)

    # quinta carta de La casa
    if continuar_juego:
        continuar_juego = turno_casa(puntos_jugador, mano_casa)

    puntos_casa = juego.contar_puntos_mano(mano_casa)
    if puntos_casa < puntos_jugador and puntos_jugador <= 21:
        print("¡Ganaste!: tienes {} puntos vs {} puntos de la casa".format(puntos_jugador,
puntos_casa))

iniciar_aplicacion()

```

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 cree que 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?

By Mario Sánchez
© Copyright Agosto de 2020.

[Departamento de Ingeniería de Sistemas y Computación](#) - Universidad de los Andes

3.2. Lógica y valores de verdad

[Print to PDF](#)

On this page

[3.2.1. Valores de verdad y proposiciones](#)

[3.2.2. Álgebra Booleana](#)

[3.2.2.1. Operaciones lógicas y tablas de verdad](#)

[3.2.2.2. Tablas de verdad](#)

[3.2.2.3. Leyes fundamentales](#)

[3.2.2.4. Leyes de De Morgan](#)

[3.2.3. Ejercicios](#)

⚠ 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.

ℹ️ Objetivo de la sección

El objetivo de esta sección es repasar conceptos fundamentales de lógica tales como el concepto de proposición, valor de verdad, conjunción, disyunción y negación. La sección presentará los conceptos desde el punto de vista de lógica matemática antes de que sean aplicados a Python en la siguiente sección.

Hasta el momento hemos trabajado en nuestros programas usando valores de tipo numéricos y con cadenas de caracteres. A partir de ahora vamos a usar también *valores de verdad*, es decir valores que sólo pueden ser verdaderos o falsos. Estos valores son también llamados valores lógicos o Booleanos (por el matemático George Boole) y se pueden operar utilizando las reglas del *álgebra Booleana*, en lugar de las reglas del álgebra elemental que usamos para valores numéricos.

En esta sección veremos cómo representar, manipular y operar con valores de verdad, haciendo un repaso rápido de lógica matemática, incluyendo las operaciones fundamentales y las leyes y propiedades que deben tenerse en cuenta para su manipulación.

3.2.1. Valores de verdad y proposiciones

Una proposición es una expresión completa que tiene un valor de verdad, es decir que es verdadera o falsa. Por ejemplo, la frase ‘Hoy está haciendo calor’ es una proposición porque expresa una idea que puede ser verdadera o falsa. Evidentemente nosotros no podemos saber si, en el momento en que el lector vea esto, estará haciendo calor o no, pero el hecho de que no lo sepamos no hace que la frase deje de ser una proposición. Por el contrario, expresiones como ‘azul’, ‘mañana’ o ‘cantando’ no son frases completas, no se puede decir que sean verdaderas o falsas y por ende no pueden ser proposiciones. En español se suelen usar también los términos *sentencia*, *afirmación* o *juicio* para referirse a una proposición.

Cuando se usan proposiciones dentro del contexto de lógica matemática, se espera que tengan un valor de verdad que, como dijimos, puede ser verdadero o falso. Dependiendo del momento en que se evalúe, una proposición podría cambiar de valor (piense en la frase ‘Hoy está haciendo calor’), pero no es posible que una proposición tenga dos valores diferentes al mismo tiempo.

Finalmente, debemos recalcar que los únicos dos valores de verdad son *verdadero* y *falso*, independientemente de cómo se representen. Por ejemplo, es usual que estos valores se representen con expresiones como *V* y *F*, *true* y *false*, *T* y *F* o incluso números como *1* y *0*. Sin embargo, en todos estos casos estamos hablando de los valores de verdad y no de expresiones de tipo numérico o de cadenas de caracteres.

3.2.2. Álgebra Booleana

El álgebra Booleana es la rama del álgebra que trabaja con proposiciones y no con valores numéricos como el álgebra elemental: en lugar de tener los números *0* y *1* y las operaciones de adición y multiplicación como elementos fundamentales, el álgebra Booleana se basa en los valores verdadero y falso y las operaciones \wedge (conjunción), \vee (disyunción) y \neg (negación). Debido a limitaciones en el formato de este libro, en lugar de usar siempre los símbolos usuales para estas operaciones, usaremos también las expresiones *and*, *or* y *not* para referirnos respectivamente a la conjunción, la disyunción y la negación.

3.2.2.1. Operaciones lógicas y tablas de verdad

Como dijimos, son tres las operaciones básicas del álgebra Booleana. El resto de operaciones, como la implicación o la equivalencia, son operaciones secundarias que se pueden construir a partir de la conjunción, disyunción y negación.

1. Negación. La negación (\neg , *not*, *no*) es la operación Booleana que toma un valor de verdad y lo convierte en el otro valor. Es decir que la negación de un valor verdadero es un valor falso, y la negación de un valor falso es un valor verdadero. La negación es una operación *unaria*, que se aplica sobre un solo operando.
2. Conjunción. La conjunción (\wedge , *and*, *y*) es una operación Booleana binaria que tiene valor verdadero sólo cuando ambos operandos tienen valor verdadero. Esto implica que cuando un operando es verdadero y el otro es falso, o cuando ambos operandos son falsos, el resultado de una conjunción es un valor falso.
3. Disyunción. La disyunción (\vee , *or*, *o*) es una operación Booleana binaria que tiene valor verdadero cuando por lo menos uno de los operandos tiene valor verdadero. Esto implica que una disyunción tiene valor falso sólo cuando ambos operandos tienen valor falso.

Veamos ahora algunos ejemplos de estas operaciones aplicadas a proposiciones sencillas en Español. Para esto lo primero que vamos a definir son tres proposiciones que identificaremos con las letras *P*, *Q* y *R*.

- P: Hoy está haciendo calor
- Q: Hoy es martes
- R: Hoy es un día de fiesta

Las siguientes son 6 expresiones lógicas basadas en las 3 proposiciones y en los operadores lógicos. Para cada expresión se presenta también una “traducción” lo más directa posible a una frase en Español.

1. Q: Hoy es martes.
2. $\neg P$: Hoy no está haciendo calor
3. $\neg Q$: Hoy no es martes
4. $P \wedge Q$: Hoy está haciendo calor y es martes
5. $Q \wedge \neg R$: Hoy es martes y no es un día de fiesta
6. $P \vee R$: Hoy está haciendo calor o es un día de fiesta

A continuación, analizamos estas expresiones una por una.

1. La primera expresión será verdadera sólo cuando sea evaluada un día que sea martes; de lo contrario, será falsa.
2. La segunda expresión será verdadera sólo cuando sea evaluada un día en que no haga calor. Esto quiere decir que la expresión $\neg P$ será verdadera sólo cuando la expresión *P* sea falsa. De igual forma, $\neg P$ será falsa sólo cuando la expresión *P* sea verdadera.
3. La tercera expresión es similar y será verdadera sólo cuando sea evaluada un día que no sea martes.
4. La cuarta expresión usa una conjunción y será verdadera sólo cuando *P* y *Q* sean verdaderas simultáneamente. Es decir, cuando sea evaluada un día en que esté haciendo calor y también sea martes. Si cualquiera de las dos partes es falsa, o si las dos son falsas, entonces la conjunción será falsa.
5. La quinta expresión usa también una conjunción, pero en este caso será verdadera sólo cuando *Q* sea verdadera y cuando $\neg R$ sea verdadero, es decir cuando *R* sea falso. En otras palabras, será verdadera sólo cuando se evalúe un martes que no sea día de fiesta. Si se evalúa cualquier otro día de la semana, o si se evalúa un martes que además sea día de fiesta, entonces la expresión será falsa.
6. La sexta expresión es la que tiene una interpretación lógica que podría estar más alejada de lo que dice la intuición. En este caso se está usando una disyunción, así que tendrá un valor verdadero cuando al menos uno de los operandos sea verdadero. Esto quiere decir que la expresión será verdadera cuando esté haciendo calor (*P*) o cuando sea un día de fiesta (*R*). Si las dos partes son verdaderas (está haciendo calor y es un día de fiesta), la expresión también será verdadera. La expresión sólo será falsa cuando las dos partes sean falsas simultáneamente, es decir cuando *P* sea falsa y *R* sea falsa, o cuando $\neg P$ sea verdadera y $\neg R$ también sea verdadera.

Expresiones similares a este último ejemplo a veces se interpretan equivocadamente como si hubiera un “o exclusivo” que sólo sería verdadero cuando una de las expresiones fuera verdadera y la otra fuera falsa. Existe un operador para expresar esta relación usualmente llamado *xor*, el cual se puede construir a partir de operaciones de conjunción, disyunción y negación.

3.2.2.1. Ejercicios

1. Escriba la “traducción” más clara posible a español de las siguientes expresiones y analice en qué casos sería verdadera y en qué casos sería falsa:
 - $Q \vee R$
 - $\neg P \wedge \neg R$
 - $Q \wedge Q$
 - $Q \wedge \neg Q$
 - $P \wedge (Q \vee R)$
 - $(P \wedge Q) \vee R$
 - $(P \wedge \neg R) \vee (\neg P \wedge R)$

2. Escriba las expresiones equivalentes a las siguientes proposiciones en español, usando los 3 identificadores P, Q y R.

- Hoy no es martes
- Hoy no es martes o hoy es martes
- Hoy está haciendo calor o es martes o es un día de fiesta
- Hoy ni es martes ni es un día de fiesta

3. Simplifique las siguientes expresiones (escríbalas de una forma más breve) si es posible:

- $Q \wedge Q$
- $Q \wedge \neg Q$
- $Q \wedge \text{Verdadero}$
- $Q \wedge \text{Falso}$
- $Q \vee \text{Verdadero}$
- $Q \vee \text{Falso}$
- $\neg Q \wedge \neg Q$
- $\neg Q \vee \neg Q$
- $\neg(Q \vee P)$
- $\neg(Q \wedge P)$
- $(P \wedge Q) \vee (P \wedge R)$

3.2.2. Tablas de verdad

Para estudiar el comportamiento de una expresión es usual que se utilice algo llamado una tabla de verdad. Esta no es más que una representación sencilla que nos permite analizar todos los valores posibles de una expresión, en función de los valores de las proposiciones simples involucradas.

Consideremos la tabla más sencilla posible, en la cual sólo tenemos la proposición P [1].

P	$\neg P$	
T	F	
F	T	

Aunque sencilla, esta tabla es interesante porque nos muestra todos los posibles valores de la expresión P y los valores correspondientes que tendría la expresión $\neg P$. También nos muestra que las dos expresiones no pueden ser verdaderas simultáneamente, sino que tienen valores opuestos. Si extendemos la tabla con un par de expresiones encontraremos dos principios muy importantes:

P	$\neg P$	$P \wedge \neg P$	$P \vee \neg P$	
T	F	F	T	
F	T	F	T	

En primer lugar, vemos que la expresión $P \wedge \neg P$ siempre es falsa. Esto quiere decir que no es posible que una proposición sea simultáneamente verdadera y falsa, lo cual usualmente se conoce como el principio de no contradicción.

En segundo lugar, vemos que la expresión $P \vee \neg P$ siempre es verdadero. Esto quiere decir que siempre bien sea una proposición o su negación tienen que ser verdaderas. Como no hay una tercera opción, eso se conoce como el principio del tercero excluido.

La siguiente tabla de verdad es un poco más compleja porque involucra dos proposiciones.

P Q $P \wedge Q$ $P \vee Q$

T	T	T	T
---	---	---	---

T	F	F	T
---	---	---	---

F	T	F	T
---	---	---	---

F	F	F	F
---	---	---	---

Acá podemos ver, de otra manera, lo que ya sabíamos sobre la conjunción y la disyunción: una conjunción es verdadera sólo cuando los dos operandos son verdaderos y una disyunción es falsa sólo cuando los dos operandos son falsos.

3.2.2.3. Leyes fundamentales

Así como en el álgebra elemental hay algunas leyes muy importantes y conocidas (comutatividad, asociatividad, distribución, precedencia de operadores, etc.), en el álgebra Booleana también hay algunas leyes importantes que se deben tener en cuenta y pueden servir para replantear expresiones de formas que sean más sencillas.

1. Comutatividad: Tanto la conjunción como la disyunción son comutativas, así que $A \vee B$ es equivalente a $B \vee A$. También es cierto que $A \wedge B$ es equivalente a $B \wedge A$ [2].
2. Asociatividad: Tanto la conjunción como la disyunción son asociativas, así que $A \vee (B \vee C)$ es equivalente a $(A \vee B) \vee C$. Además, $A \wedge (B \wedge C)$ es equivalente a $(A \wedge B) \wedge C$ [3].
3. Distribución: en el álgebra Booleana, la conjunción distribuye sobre la disyunción y viceversa [4]. Esto quiere decir que:
 - $A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C)$
 - $A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$
4. Identidad de la conjunción: si se hace una conjunción con el valor verdadero, el resultado es el mismo. Es decir que $A \wedge \text{Verdadero} \equiv A$ [5].
5. Identidad de la disyunción: si se hace una disyunción con el valor falso, el resultado es el mismo. Es decir que $A \vee \text{Falso} \equiv A$.
6. Dominación de la conjunción: si se hace una conjunción con el valor falso, el resultado es falso. Es decir que $A \wedge \text{Falso} \equiv \text{Falso}$ [6].
7. Dominación de la disyunción: si se hace una disyunción con el valor verdadero, el resultado es verdadero. Es decir que $A \vee \text{Verdadero} \equiv \text{Verdadero}$.

Finalmente, proponemos la siguiente equivalencia que, aunque no tiene un nombre común, es de uso extremadamente frecuente y de mucha utilidad cuando se está empezando a programar:

- Negación y equivalencia a falso: es lo mismo decir que una proposición tiene un valor falso que decir que su negación tiene un valor positivo. Es decir que $A \equiv \text{Falso}$ es lo mismo que decir $\neg A \equiv \text{Verdadero}$. Más aún, $B \equiv \text{Verdadero}$ es lo mismo que decir B , así que $A \equiv \text{Falso}$ se puede reescribir como $\neg A$.

3.2.2.4. Leyes de De Morgan

Finalizamos este repaso de lógica con dos teoremas muy importantes que se deben tener muy en cuenta cuando se estén reescribiendo operaciones lógicas.

Las leyes o teoremas de De Morgan dicen que:

- $\neg (P \wedge Q) \equiv \neg P \vee \neg Q$
- $\neg (P \vee Q) \equiv \neg P \wedge \neg Q$

Volviendo a las proposiciones de ejemplo que usamos anteriormente, la expresión $\neg (P \wedge Q)$ podría “traducirse” como “no es cierto que (hoy está haciendo calor y hoy es martes)”. Hemos agregado los paréntesis para hacer notar que las palabras “no es cierto que” hacen referencia a las dos cláusulas subordinadas.

Sabemos que la expresión completa será verdadera sólo cuando la parte que está entre paréntesis sea falsa para que la negación invierta su valor. Es decir que la expresión completa será verdadera cuando sea falso que “hoy está haciendo calor y hoy es martes”. En este caso tenemos una conjunción, así que la operación será verdadera

sólo cuando ambas sean falsas. Como en este caso nos interesa que sea falsa, decimos que la operación de conjunción será falsa cuando cualquiera de las dos partes sea falsa, es decir cuando la primera parte sea falsa o la segunda parte sea falsa. Volviendo a la expresión completa, encontramos que la expresión será verdadera cuando la proposición P sea falsa o cuando la proposición Q sea falsa.

Esto quiere decir que la expresión original también podría haberse traducido de forma equivalente como “no es cierto que hoy esté haciendo calor o no es cierto que hoy sea martes”.

La segunda propiedad es análoga a la primera.

3.2.3. Ejercicios

1. Construya una tabla de verdad para verificar cada una de las leyes de De Morgan. Para la primera, necesitará una columna para P, una para Q, una para $\neg(P \wedge Q)$ y una para $\neg P \vee \neg Q$. Debería encontrar exactamente los mismos valores en las últimas dos columnas. Para facilitar el trabajo, también le recomendamos agregar una columna con los valores de las expresiones $\neg P$, $\neg Q$, $P \wedge Q$ y $P \vee Q$.
2. En un ejercicio anterior se le pidió simplificar unas expresiones. Vuelva a hacer el ejercicio construyendo una tabla de verdad para cada caso y aplicando las leyes que se vieron al final de esta sección.
3. Como vimos, el álgebra booleana se define en términos de dos valores y tres operaciones (conjunción, disyunción y negación). Otras operaciones lógicas pueden definirse como composiciones de las operaciones básicas. Por ejemplo, la operación xor, que es verdadera sólo cuando exactamente uno de los operandos es verdadero, se puede definir como: $P \text{ xor } Q \equiv (P \wedge \neg Q) \vee (Q \wedge \neg P)$. Defina cada una de las siguientes operaciones y para cada una construya una tabla de verdad que muestre su comportamiento y le sirva para verificar su definición.
 - xor (or exclusivo): es verdadero cuando exactamente un operando es verdadero.
 - == (equivalencia): es verdadero cuando los operandos tienen el mismo valor.
 - != (desigualdad): es verdadero cuando los operandos tienen valores diferentes.
 - nand: es falso sólo cuando los dos operandos son verdaderos.
 - nor: es verdadero sólo cuando los dos operandos son falsos.

-
- [1] Para mantener las tablas relativamente pequeñas, usaremos T para valores verdaderos y F para valores falsos.
- [2] En el álgebra elemental tanto la adición como la multiplicación son comutativas.
- [3] En el álgebra elemental tanto la adición como la multiplicación son asociativas. Es decir que $a + (b + c)$ es igual a $(a + b) + c$.
- [4] El álgebra elemental, la multiplicación distribuye sobre la adición, pero no al contrario. Es decir que $a \times (b+c)=a \times b + a \times c$, pero la ecuación no se mantiene si se intercambian las operaciones.
- [5] Esto es equivalente a multiplicar por 1 o a sumar 0.
- [6] Esto es equivalente a multiplicar por 0. No existe algo similar para la adición.

By Mario Sánchez
© Copyright Agosto de 2020.

[Departamento de Ingeniería de Sistemas y Computación](#) - Universidad de los Andes

3.3. Valores de verdad en Python

[Print to PDF](#)

On this page

[3.3.1. El tipo bool en Python](#)

[3.3.1.1. Conversiones](#)

[3.3.1.2. Funciones que retornan valores de verdad](#)

[3.3.2. Operadores Booleanos](#)

[3.3.2.1. Conjunction \(and\)](#)

[3.3.2.2. Disyunción \(or\)](#)

[3.3.2.3. Negación \(not\)](#)

[3.3.3. Operadores relacionales](#)

[3.3.3.1. Operadores de orden](#)

[3.3.3.2. Operadores de igualdad](#)

[3.3.3.3. Operadores de identidad](#)

[3.3.4. Ejercicios](#)

[3.3.5. Más allá de Python](#)

⚠ 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.

ℹ️ Objetivo de la sección

El objetivo de esta sección es mostrar cómo se representan y operan valores de verdad dentro de un lenguaje de programación. Los conceptos que se estudian en esta sección son aplicables directamente a la mayoría de lenguajes de programación.

En la sección pasada trabajamos con valores de verdad y con proposiciones. En esta sección veremos que Python tiene un tipo de datos específico para este tipo de valores (`bool`) y que permite también la construcción de expresiones lógicas usando conjunciones, disyunciones y negaciones. También aprenderemos a escribir expresiones lógicas usando operadores relacionales sobre valores de todos los tipos que conocemos hasta el momento (números, cadenas y valores de verdad). En la próxima sección usaremos todos estos conceptos para construir expresiones condicionales.

3.3.1. El tipo bool en Python

En el nivel 1 nos encontramos ya con los tipos que Python utiliza para manejar valores numéricos (`int` y `float`) y cadenas de caracteres (`str`). Ahora vamos a introducir el tipo `bool`, que se utiliza para representar valores de verdad y puede utilizarse en variables, parámetros, llamados de funciones y otras expresiones.

En comparación con los tipos que ya conocíamos, `bool` es mucho más sencillo puesto que sólo hay dos literales: `True`, que se utiliza para expresar un valor verdadero, y `False`, que se usa para expresar un valor falso. Ponga atención al uso de minúsculas y mayúsculas, puesto que los dos literales se deben escribir exactamente así.

También note que no es lo mismo '`True`' que `True`. El primer valor es una cadena de caracteres, mientras que el segundo es un valor de verdad:

```
>>> type('True')
<class 'str'>
>>> type(True)
<class 'bool'>
```

3.3.1.1. Conversiones

Al igual que con los otros tipos, existe la función `bool` que permite convertir de otros tipos a booleano. Sin embargo, no es muy recomendable utilizar esta función puesto que su uso muchas veces lleva a resultados inesperados.

Por ejemplo, las siguientes expresiones tienen valor equivalente a `True`:

- `bool('True')`
- `bool('true')`
- `bool('cadena')`
- `bool(5)`
- `bool(-5)`

También tienen valor equivalente a `True` las siguientes expresiones, aunque naturalmente se podría esperar lo contrario:

- `bool('False')`
- `bool('false')`
- `bool('')`

Por otro lado, las siguientes expresiones tienen todos valor equivalente a `False`:

- `bool(0)`
- `bool(-0)`
- `bool('')`

Como dijimos antes, es mejor limitarse a usar los literales `True` y `False` que aprenderse todas las reglas que explican estas conversiones. Todo lo que puede hacerse usando esas conversiones puede hacerse también sin tener que recurrir a ellas [1].

3.3.1.2. Funciones que retornan valores de verdad

El tipo `bool` también puede usarse para expresar el tipo de retorno de una función. Por ejemplo, las siguientes son dos funciones que retornan un valor de tipo `bool`: siempre que la primera función sea invocada, el resultado será un valor verdadero, mientras que para la segunda función el valor siempre será falso.

```
def f(x: int)->bool:
    print('f:', x)
    return True

def g(x: int)->bool:
    print('g:', x)
    return False
```

Estas dos funciones no parecen muy interesantes porque siempre retornan el mismo valor, pero más adelante en este capítulo las utilizaremos para una importante demostración.

3.3.2. Operadores Booleanos

A continuación, estudiaremos los operadores que Python ofrece para realizar las operaciones Booleanas de conjunción, disyunción y negación.

Algo importante que se debe tener en cuenta es que, aunque las operaciones son commutativas, el orden en que se escriban los operandos tiene un efecto sobre la forma en la que se evalúan las expresiones. Para cada operador explicaremos esto más en detalle.

3.3.2.1. Conjunción (and)

La operación lógica de conjunción se expresa en Python usando la palabra reservada `and`. Cuando se tiene en un programa Python una expresión de la forma `p and q` se sabe entonces que tendrá un valor verdadero sólo si la expresión `p` y la expresión `q` tienen simultáneamente un valor verdadero. Además, el operador `and` es asociativo, así que se pueden escribir expresiones como `w and x and y and z` sin tener que usar paréntesis y sin riesgos de que cambie el resultado.

Ahora bien, hay una pequeña pero muy significativa diferencia entre el operador `and` y la operación lógica de conjunción. Para estudiarla, tomemos como ejemplo la expresión `w and x and y and z`. Si se tratara de una expresión lógica, la podríamos haber escrito en cualquier orden y el resultado sería siempre el mismo debido a la commutatividad de la conjunción. Es decir que para nosotros sería exactamente lo mismo escribir `z and x and w and y`.

En el caso de la expresión Python entran a jugar también consideraciones prácticas. Por ejemplo, suponga que `x` es un valor de verdad extremadamente difícil de calcular [2] y suponga también que sabemos que `w` es una expresión falsa. No tendría sentido buscar el valor de `x` puesto que sabríamos de antemano que la expresión completa sería falsa.

La diferencia entonces entre el operador `and` de Python y la operación de conjunción es que Python se aprovecha de las propiedades de identidad y dominancia de la conjunción ($p \wedge \text{Verdadero} = p$ y $p \wedge \text{Falso} = \text{Falso}$) para evitar calcular más términos de los necesarios.

Esto quiere decir que cuando se tiene una operación de conjunción primero se evalúa sólo el primer término: si su valor es falso, se sabe que toda la expresión será falsa y no tiene sentido seguir avanzando. Pero si el valor del término es verdadero, entonces se puede concluir que el valor de la expresión será igual al valor de la parte restante de la expresión.

Apliquemos esto a nuestro ejemplo `w and x and y and z`:

- Python primero revisará el valor de `w`. Si es falso, Python inmediatamente dirá que la expresión completa es falsa. De lo contrario, el valor de la expresión completa será el valor de la expresión `x and y and z`.
- Si no hemos terminado, Python revisará el valor de `x`. Si es falso, Python inmediatamente dirá que la expresión completa es falsa. De lo contrario, el valor de la expresión completa será el valor de la expresión `y and z`.

- Si no hemos terminado, Python revisará el valor de `y`. Si es falso, Python inmediatamente dirá que la expresión completa es falsa. De lo contrario, el valor de la expresión completa será el valor de la expresión `z`.
- Si no hemos terminado, Python revisará el valor de `z` y como no hay más operaciones le asignará a la expresión el valor de `z`.

Veamos ahora esto mismo con un ejemplo un poco más elaborado en el que vamos a tener nuestras dos funciones que siempre retornan el mismo valor: la función `f` siempre retornará el valor `True` mientras que la función `g` siempre retornará el valor `False`. Además, cada función dejará una traza en la consola para que podamos ver en qué orden fueron llamadas.

```
def f(x: int)->bool:
    print('f:', x)
    return True

def g(x: int)->bool:
    print('g:', x)
    return False

print("Caso 1 - f and f and f :")
print(f(1) and f(2) and f(3))

print("Caso 2 - f and f and g :")
print(f(1) and f(2) and g(3))

print("Caso 3 - f and g and g :")
print(f(1) and g(2) and g(3))

print("Caso 4 - g and g and g :")
print(g(1) and g(2) and g(3))
```

Las instrucciones que se encuentran al final del programa se encargarán de evaluar una expresión basada en invocaciones a `f` y `g` e imprimirán el resultado. Veamos ahora el resultado de ejecutar el programa:

```
Caso 1 - f and f and f :
f: 1
f: 2
f: 3
True
Caso 2 - f and f and g :
f: 1
f: 2
g: 3
False
Caso 3 - f and g and g :
f: 1
g: 2
False
Caso 4 - g and g and g :
g: 1
False
```

Lo que vemos en el caso 1 es que la función `f` se invoca tres veces y deja traza de las tres invocaciones (podemos ver que se llamó a la función `f` y el valor que se le asignó al parámetro `x`). También vemos que el resultado que se imprime es `True` y corresponde al valor de la operación de conjunción.

En el caso 2 lo que vemos es que se invocó dos veces la función `f` y luego se invocó la función `g`. El resultado de esta expresión es `False`.

El tercer caso es mucho más interesante: podemos ver que la función `f` se invocó una vez, pero la función `g` sólo se invocó una vez en lugar de dos veces. Esto se debe a que, una vez se encontró el valor `False` que retornó la función `g`, ya se conocía el valor de la expresión completa y no tenía sentido continuar evaluando los otros términos.

En el cuarto caso nos encontramos el mismo comportamiento: sólo se evaluó el primer término y, como tenía valor falso, hizo que toda la expresión fuera falsa sin tener que evaluar el resto de los términos.

3.3.2.2. Disyunción (or)

La operación lógica de disyunción se expresa en Python usando la palabra reservada `or`. Cuando se tiene en un programa Python una expresión de la forma `p or q` se sabe entonces que tendrá un valor falso sólo si la expresión `p` y la expresión `q` tienen simultáneamente un valor falso. Además, el operador `or` es asociativo, así que se pueden escribir expresiones como `w or x or y or z` sin tener que usar paréntesis y sin riesgos de que cambie el resultado.

Al igual que con la operación `and`, Python también busca eficiencias en la evaluación de expresiones que usen en este operador. Esto se logra identificando inmediatamente que el resultado de una operación `or` entre el valor verdadero y cualquier valor, siempre será verdadero. Es decir que, si el primer operando de una operación que use el operador `or` es verdadero, entonces el valor de la operación será verdadero y no será necesario evaluar los otros términos.

Veamos esto agregándole unas instrucciones adicionales a nuestro programa anterior:

```
print("Caso 1 - f or f or f :")
print(f(1) or f(2) or f(3))

print("Caso 2 - f or f or g :")
print(f(1) or f(2) or g(3))

print("Caso 3 - g or f or g :")
print(g(1) or f(2) or g(3))

print("Caso 4 - g or g or g :")
print(g(1) or g(2) or g(3))
```

Ahora se evaluarán expresiones basadas en disyunciones y tendremos un comportamiento muy diferente al del caso anterior:

```
Caso 1 - f or f or f :
f: 1
True
Caso 2 - f or f or g :
f: 1
True
Caso 3 - g or f or g :
g: 1
f: 2
True
Caso 4 - g or g or g :
g: 1
g: 2
g: 3
False
```

En el primer caso, vemos que Python inmediatamente identifica que el primer término tiene valor verdadero. Esto implica que la expresión completa tendrá valor verdadero y la evaluación termina sin tener que evaluar los otros términos.

En el segundo caso pasa algo exactamente igual: a pesar de que los otros términos tienen valores diferentes, es suficiente con evaluar el primer término para saber cuál será el valor de toda la expresión.

En el tercer caso lo que vemos es que se evalúa el primer término que, como sabemos por la definición de la función, es falso. Python tiene entonces que recurrir al segundo término, que en este caso es verdadero. Esto hace que la expresión completa sea verdadera y no sea necesario seguir con el resto de la evaluación.

Sólo en el cuarto caso se evalúan los tres términos: el primero es falso, obligando a la evaluación del segundo, que también es falso. Finalmente se evalúa el tercer término y, como también es falso se concluye que la expresión completa era falsa.

3.3.2.3. Negación (not)

El tercer operador lógico en Python es la negación, la cual se expresa con la palabra reservada `not`. Como se trata de un operador unario se debe anteponer al operando. Es decir que se usa igual que como se usa el signo `-` para convertir a un número en su negativo (por ejemplo, `5`, `-5` y `-(-5)`).

El operador `not` no tiene ninguna diferencia con la operación lógica de negación y tiene una precedencia que es mayor a la de la conjunción y a la de la disyunción. Es decir que `not a and b` siempre será equivalente a `(not a) and b` y que `a and not b` siempre será equivalente a `a and (not b)`.

Veamos esto en un ejemplo:

```
print("Caso 1:", not f(1) and f(2))
print(not f(1) and f(2))
print("Caso 2:", not g(1) and f(2))
print(not g(1) and f(2))
```

El resultado de la ejecución de este programa se muestra a continuación:

```
Caso 1: not f(1) and f(2)
f: 1
False
Caso 2: not g(1) and f(2)
g: 1
f: 2
True
```

En el primer caso vemos que se evaluó la función `f` y se obtuvo necesariamente el valor `True`. A continuación, se aplicó el operador de negación y se obtuvo el valor `False`. En ese punto Python pudo descubrir que la expresión completa iba a tener valor `False` y la evaluación terminó con ese resultado.

En el segundo caso vemos que, por el contrario, el primer término de la conjunción tiene valor `True` porque primero se evaluó `g` y luego se aplicó la negación. Después de esto se hizo la evaluación del segundo término y se llegó al valor final: `True`.

Note que en ambos casos la evaluación de la operación de negación se hizo antes de la evaluación de la operación de conjunción.

[^ sobre]: En realidad, en Python se pueden usar otros literales para expresar valores verdaderos y falsos, pero en general es mucho más claro cuando se usan los literales `True` y `False`. Por ejemplo, las siguientes dos expresiones tienen valores que podríamos llamar *desconcertantes*: 1. `not -66 and 'a'` 2. `not -66 and 'a'`. La primera expresión tiene valor `False`, mientras que la segunda tiene valor `'a'`. Aunque esos resultados no son un error y están perfectamente justificados en la especificación de Python, debería ser mucho más fácil deducir el valor de una expresión sin tener que conocer detalles de implementación relativamente oscuros.

3.3.2.3.1. Ejercicios

1. ¿Cuál es el valor de las siguientes expresiones Python basadas en conjunciones? ¿Cuáles de las reglas y propiedades que se estudiaron en la sección anterior se aplican en cada caso?

- `False and p`
- `True and p`
- `p and True`
- `p and False`
- `p and q and r`

2. ¿Cuál es el valor de las siguientes expresiones Python basadas en disyunciones? ¿Cuáles de las reglas y propiedades que se estudiaron en la sección anterior se aplican en cada caso?

- `False or p`
- `True or p`
- `p or True`
- `p or False`
- `p or q or r`

3. ¿Cuál es el valor de las siguientes expresiones Python basadas en negaciones?

- `not True`
- `not not True`
- `not not not False`

4. ¿Cuáles de las reglas y propiedades que se estudiaron en la sección anterior se podrían aplicar en cada una de las siguientes expresiones?

- `not p and not q`
- `(p and q) or (q and r)`
- `p and (q or r)`
- `p or (q and r)`

3.3.3. Operadores relacionales

Hasta este momento todas las operaciones que hemos visto cuyo resultado es un valor de verdad han utilizado otros valores de verdad como operandos. Los operadores relacionales que veremos a continuación nos permitirán hacer comparaciones entre diferentes valores con el fin de obtener un valor de verdad.

La siguiente tabla resume los operadores disponibles en Python para hacer comparaciones entre valores. Estos operadores pueden aplicarse a cualquier tipo de valor, aunque no necesariamente el resultado tendrá un sentido muy evidente. Por ejemplo, la expresión `True < False` es falsa, pero no es fácil imaginarse una razón para comparar dos valores booleanos usando ese operador.

Operador	Significado	Ejemplo	Resultado
<	Es menor que ...	4 < 7	True
<=	Es menor o igual que ...	"Ab" <= "ab"	True
>	Es mayor que ...	4.5 > 7.1	False
>=	Es mayor o igual que ...	'1A' >= 'A1'	False
==	Es igual a ...	"abc" == "ab" + "c"	True
!=	Es diferente de ...	4 != 7	True
is	Dos <i>objetos</i> son el mismo	4 is 7	False
is not	Dos <i>objetos</i> no son el mismo	4 is not 7	True

3.3.3.1. Operadores de orden

Los primeros cuatro operadores tienen un comportamiento intuitivo cuando se aplican sobre valores numéricos ([int](#) y [float](#)). Cuando se aplican sobre cadenas de caracteres, la comparación se hace de forma lexicográfica (como se organizarían las palabras en un diccionario). De esta manera, la cadena "[holo](#)" debería ser *menor* que la cadena [mundo](#). Sin embargo, en este sistema las mayúsculas siempre son *menores* que las minúsculas, así que la expresión '[Z](#)' < '[a](#)' siempre será verdadera. Por su parte, los números son *menores* que las mayúsculas, y algunos símbolos son menores que los números [3]. Si se quiere hacer una comparación más sencilla se pueden convertir las cadenas a letras minúsculas o mayúsculas usando uno de los mecanismos que veremos en una de las siguientes secciones.

Si se intentan usar estos 4 operadores entre valores de tipos diferentes se presentará un error similar al siguiente:

```
TypeError: '<' not supported between instances of 'int' and 'str'
```

Las siguientes funciones nos servirán para ilustrar algunas de las ideas que acabamos de presentar.

```
def es_positivo(x: int)->bool:
    positivo = x > 0
    return positivo

def es_negativo(x: int)->bool:
    negativo = x < 0
    return negativo

def es_cero(x: int)->bool:
    positivo = es_positivo(x)
    negativo = es_negativo(x)
    return not positivo and not negativo

def ordenadas(antes: str, despues: str)->bool:
    """
    Revisa si dos cadenas en un diccionario están ordenadas Lxicográficamente
    Parámetros:
        antes (str): una cadena que está antes que la otra en un diccionario
        despues (str): una cadena que está después que la otra en un diccionario
    Retorno:
        (bool): Indica si las cadenas estaban ordenadas.
        El resultado será verdadero si la cadena 'antes' tiene que ir
        antes que la cadena 'despues' en orden Lxicográfico.
        El resultado será falso de lo contrario.
    """
    estan_ordenadas = antes < despues
    return estan_ordenadas
```

La primera función revisa si el valor que se pasa en el parámetro `x` es mayor que 0. En caso de que esto sea cierto, en la variable `positivo` debería quedar el valor `True` y este valor sería retornado por la función. En caso contrario (es decir, si `x <= 0`), la función retorna `False`.

La segunda función hace algo análogo, pero para ver si el valor que se pasa como parámetro es negativo.

La tercera función combina las dos funciones anteriores para ver si el valor `x` es cero: si el número *no* es positivo y *no* es negativo, entonces tiene que ser el valor `0`.

La cuarta función utiliza las comparaciones lexicográficas para ver si dos palabras extraídas de un diccionario estaban correctamente ordenadas. En la variable `estan_ordenadas` queda el resultado de comparar las palabras lexicográficamente, y este valor posteriormente se retorna.

3.3.3.2. Operadores de igualdad

Los siguientes operadores (`==` y `!=`) sirven para establecer si dos valores son iguales o no.

Atención: no confunda el operador `==` con la instrucción de asignación `=`. En el primer caso el resultado será un valor de verdad mientras que en el segundo caso se modificará el valor de una variable.

En el ejemplo de la tabla se puede apreciar un caso muy interesante: a la derecha del operador `==` tenemos la cadena `"abc"`, mientras que a la derecha tenemos la expresión `"ab" + "c"`. Por lo que aprendimos en el capítulo anterior, sabemos que el valor de la parte de la derecha será la concatenación de las dos cadenas, así que será `abc`. Como las dos cadenas son iguales (tienen los mismos caracteres, en el mismo orden), el valor de la expresión será verdadero.

Por otro lado, el operador `!=` sirve para comparar dos valores y saber si son diferentes. Esto es equivalente a utilizar el operador de igualdad y luego negar el resultado.

A continuación, presentamos la definición de 3 funciones que nos servirán para ilustrar el uso de los operadores que acabamos de introducir.

```
def es_par(x: int)->bool:
    # En la siguiente Línea sobran Los paréntesis,
    # pero si incluyeron para hacer más claro el código
    residuo_cero = (x % 2 == 0)
    return residuo_cero

def no_es_7_v1(x: int)->bool:
    # En la siguiente Línea sobran Los paréntesis,
    # pero si incluyeron para hacer más claro el código
    x_es_7 = (x == 7)
    return not x_es_7

def no_es_7_v2(x: int)->bool:
    return x != 7
```

La primera función verifica si un número es par. Para eso, calcula el residuo del número módulo 2 y lo compara con 0: si son iguales, significa que el número era par y la función retorna el valor `True`; si son diferentes, la función retorna el valor `False`.

La siguiente función es muy parecida, aunque en este caso lo que se retorna es la negación de la variable `x_es_7`. En este caso queríamos mostrar que es posible negar un valor justo antes de retornarlo.

La tercera función hace lo mismo que la segunda (dice si un número NO es 7), pero lo hace en la misma línea en la cual hace el retorno. Si analizamos con detenimiento esta función, veremos que lo primero que se hace es comparar el valor del parámetro `x` con el número `7` para ver si son diferentes (verdadero si son diferentes, falso si son iguales). El resultado de esta comparación es inmediatamente retornado y la función termina su ejecución.

Atención: Tenga mucho cuidado cuando haga comparaciones con números decimales. Mientras que las comparaciones de enteros usualmente no tienen problemas, las comparaciones entre decimales pueden tener inconvenientes por culpa de errores en la precisión de los cálculos en Python [4]. Por ejemplo, el valor de evaluar la expresión `0.2 == 1.2 - 1.0` es, sorprendentemente, `False`. Para entender por qué, se puede evaluar la parte derecha en el intérprete de Python, obteniendo el valor `0.1999999999999996` que, aunque es muy cercano a `0.2`, no es idéntico.

Para evitar este problema, la recomendación general es que, cuando se vayan a comparar valores numéricos con decimales se defina una precisión y se revisen los intervalos, como en el siguiente ejemplo:

```
epsilon = 0.005
iguales = (0.2 > (1.2 - 1.0) - epsilon) and (0.2 < (1.2 - 1.0) + epsilon)
```

En este caso, en lugar de comparar `0.2` con el resultado de la resta `1.2 - 1.0` estamos mirando que esté dentro del intervalo que va desde `1.2 - 1.0 - epsilon` hasta `1.2 - 1.0 + epsilon`. El valor de `epsilon` se puede definir tan grande o tan pequeño como sea conveniente para el problema que se esté resolviendo.

3.3.3. Operadores de identidad

Finalmente, los operadores `is` y `is not` sirven para revisar que dos valores no sólo sean iguales, sino que también sean el mismo. Como en este libro no vamos a ocuparnos mucho de objetos, no ahondaremos en la distinción entre `==` e `is`.

Más adelante explicaremos en qué casos tiene sentido utilizar el operador `is not` en el contexto de la expresión `is not None`.

3.3.4. Ejercicios [5]

1. Escriba una función que dada la edad de una persona indique si puede manejar (tiene que tener al menos 16 años)
2. Escriba una función que dada la altura en metros y el peso en kilogramos de un adulto diga si está dentro de los rangos típicamente considerados saludables. Para esto debe usar el Índice de Masa Corporal (BMI), que se calcula como $\text{peso}/\text{altura}^2$ ². Un adulto se considera que tiene sobrepeso cuando su BMI es mayor o igual a 25. Un adulto se considera que está bajo de peso cuando su BMI es menor a 18.5.
3. Escriba una función que dados dos números diga si el primero es divisible por el segundo.
4. Queremos saber si una persona tiene el dinero suficiente para pagar la cuenta en un restaurante dados los siguientes parámetros: la cantidad de dinero que tiene la persona, el valor de la cuenta, si la persona va a dejar propina o no. La propina corresponde al 10% del valor de la cuenta.

3.3.5. Más allá de Python

Al igual que Python, muchos lenguajes utilizan tipos especiales para representar valores de verdad, pero incluyen también convenciones tales como usar `0` para falso y cualquier otro número para verdadero. Por ejemplo, en PHP existen los literales ‘true’ y ‘false’ (escritos sin importar el uso de mayúsculas o minúsculas), pero también tienen un valor falso los números `0`, `0.0`, `-0` y `-0.0`, las cadenas vacías, la cadena “`0`”, un arreglo sin elementos, y el tipo `NULL`. Algo similar sucede en C y C++. Por el contrario, en Java los únicos literales para valores booleanos son ‘true’ y ‘false’. Como se dijo en la nota sobre el caso de Python, en general es mucho mejor que el código sea claro y se usen los valores booleanos de forma explícita en lugar de depender de conversiones.

La explicación que se presentó sobre el orden en el que se evalúan los términos de una conjunción o una disyunción en general aplica para todos los lenguajes de programación. C, C++, Java, PHP y JavaScript, entre muchos otros, implementan ideas similares.

La distinción entre `son iguales` y `son el mismo` es sutil y se presta para confusiones en diferentes lenguajes. Por ejemplo, en JavaScript existen los operadores `==` y `===` para diferenciar entre los todos tipos de comparación, mientras que en Java el operador `==` representa uno o el otro dependiendo de qué se esté comparando. Más aún, en Java los objetos de tipo String se manejan de forma diferente dependiendo de en qué momento se les asigne su valor, haciendo que a veces dos cadenas iguales sean también la misma y a veces no lo sean. Entender con absoluta claridad este punto para el lenguaje de programación que esté usando le evitará muchísimos dolores de cabeza.

- [1] Acá hemos incluido sólo algunos ejemplos de las conversiones que se pueden hacer. Hizo falta incluir ejemplos basados en números complejos, valores nulos (`None`), listas, diccionarios, conjuntos y en general todos los tipos de secuencias y colecciones.
- [2] Más adelante en este libro veremos muchos ejemplos de funciones y valores que toman tiempos largos para ser calculados.
- [3] Para entender bien el orden lexicográfico se debe revisar la tabla ASCII que se discutió en el nivel 1 y entender que lo que el resultado de la comparación es el resultado de comparar los números de cada carácter en la tabla ASCII. De esta forma, el carácter ‘k’ (#107) va después del ‘R’ (#82), que va después del ‘;’ (#59), que va después del ‘4’ (#52), que va después del ‘&’ (#38).
- [4] Este problema no es exclusivo de Python. Casi todos los lenguajes sacrifican precisión en los cálculos para ganar un poco de eficiencia y mejorar el uso de la memoria.

[5] Si usted ha programado antes o si ya leyó las siguientes secciones podría pensar que es necesario usar condicionales para resolverlos. La realidad es que todos estos ejercicios pueden resolverse usando sólo lo que se estudió en este capítulo.

By Mario Sánchez
© Copyright Agosto de 2020.

[Departamento de Ingeniería de Sistemas y Computación](#) - Universidad de los Andes

3.4. Instrucciones condicionales

[Print to PDF](#)

On this page

[3.4.1. Motivación](#)

[3.4.2. Condicionales en Python: IF-ELSE](#)

[3.4.2.1. Alternativas](#)

[3.4.2.2. Ejercicios](#)

[3.4.3. Condicionales en Python: IF-ELIF-ELSE](#)

[3.4.3.1. Condicionales consecutivos](#)

[3.4.3.2. Ejercicios](#)

[3.4.4. Aspectos metodológicos](#)

[3.4.4.1. Condicionales anidados](#)

[3.4.4.2. Returns](#)

[3.4.4.3. Expresiones vs. condicionales](#)

[3.4.5. Caso de estudio: el mayor de 4 números](#)

[3.4.5.1. Enunciado del problema](#)

[3.4.5.2. Solución 1: múltiples returns](#)

[3.4.5.3. Solución 2: un solo return](#)

[3.4.5.4. Solución 3: aproximación algorítmica](#)

[3.4.6. Ejercicios](#)

[3.4.7. Más allá de Python](#)

A 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.

I Objetivo de la sección

El objetivo de esta sección es introducir las instrucciones condicionales, mostrando por qué son necesarias, qué significan y cómo pueden utilizarse en Python.

Entre el nivel 1 y lo que llevamos hasta este momento del nivel 2, hemos estudiado tipos de datos (int, str, float, bool), operadores (aritméticos, sobre cadenas, lógicos), expresiones y variables, y un conjunto reducido de instrucciones que nos permiten:

- Asignarle un valor a una variable
- Definir una nueva función
- Invocar una función con unos argumentos
- Retornar un valor al terminar una función
- Importar un módulo existente

Aunque no parece mucho, este reducido conjunto de elementos son la base para construir cualquier tipo de programas.

En esta sección vamos a introducir un nuevo tipo de instrucción, los condicionales, con el cual incrementaremos enormemente el potencial de problemas que podemos solucionar y de programas que podemos construir.

3.4.1. Motivación

La gran limitación que tienen los programas que podemos construir hasta el momento es que siempre van a ejecutar las mismas instrucciones. Esto implica que no pueden tomar ninguna decisión y que incluso funcionalidades pequeñas son imposibles de implementar.

Tomemos como ejemplo la funcionalidad de verificar si la contraseña introducida por un usuario es correcta o no: si es correcta, quisiéramos informarle al usuario que sí es correcta; si no es correcta, quisiéramos decirle que no es correcta y que debe volver a intentarlo. En la sección anterior aprendimos a comparar cadenas de caracteres, así que ya podríamos saber si la contraseña es correcta o no:

```
contraseña_correcta = contraseña_almacenada == contraseña_ingresada
```

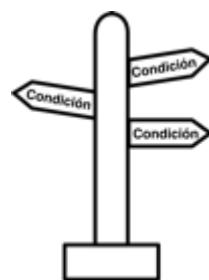
Lo que no podemos hacer, sin usar instrucciones condicionales, es *calcular* el mensaje correcto para mostrarle al usuario con base en el valor de `contraseña_correcta`.

Consideremos ahora otro ejemplo en el que queremos calcular el valor de una factura:

```
valor_producto = 14000
impuestos = valor_producto * 0.19
valor_factura = valor_producto + impuestos
```

¿Qué pasa si no todos los productos tienen que pagar impuestos? ¿Qué pasa si el porcentaje de impuestos es diferente dependiendo del valor del producto (ej. 19% para productos de más de 10000 y 10% para productos de menos)? Para lograr representar estas situaciones también necesitamos usar instrucciones condicionales.

Cuando una persona recorre un camino, es usual que llegue a un punto en el que tiene que detenerse, analizar algo y tomar una decisión sobre qué camino seguir. Eso es precisamente lo que nos van a permitir modelar las instrucciones condicionales: marcar un punto en el programa en el que se tiene que evaluar una condición y continuar por un camino dependiendo del resultado de la evaluación.



3.4.2. Condicionales en Python: IF-ELSE

Una instrucción condicional es una instrucción cuya ejecución depende del valor que resulte de evaluar una expresión booleana. Esta evaluación se hace durante la ejecución de un programa, así que cuando estamos programando nosotros no podemos saber a ciencia cierta qué es lo que va a ejecutarse. Además, ejecuciones sucesivas del mismo programa no necesariamente van a comportarse de la misma manera.

En Python, las instrucciones condicionales se usan utilizando la siguiente estructura:

```
if expresion_booleana:
    cuerpo_if
else:
    cuerpo_else
```

- `expresion_booleana`: es una expresión que tiene que tener un valor de verdad (una operación relacional, una variable booleana, un llamado a una función booleana, etc.). Note los dos puntos (:) al final de la expresión.
- `cuerpo_if`: son las instrucciones que se van a ejecutar si la expresión booleana es verdadera en el momento de la evaluación. El cuerpo del IF debe ir indentado como el cuerpo de una función.
- `cuerpo_else`: son las instrucciones que se van a ejecutar si la expresión booleana es falsa en el momento de la evaluación. El cuerpo del ELSE también debe ir indentado.

Veamos ahora cómo podemos usar esta estructura usando el ejemplo de la verificación de contraseña:

```
contraseña_almacenada = "secreto"
contraseña_ingresada = input("Ingrese su contraseña:")

if contraseña_almacenada == contraseña_ingresada:
    mensaje = "Su contraseña es correcta"
else:
    mensaje = "La contraseña es incorrecta"

print(mensaje)
```

En primer lugar, tenemos que notar que este programa tiene sólo cuatro instrucciones:

1. una asignación,
2. un llamado a una función (`input`),
3. una instrucción condicional y
4. una invocación a una función (`print`).

Como veremos, la instrucción condicional tiene varios elementos por dentro, pero deberíamos considerarla una sola instrucción.

En la primera instrucción se hace una asignación y se almacena en una variable la cadena '`'secreto'`' para que ese sea la contraseña contra la que se va a hacer la verificación.

En la segunda instrucción se hace una invocación a la función `input` y se espera a que el usuario ingrese un valor. Ese valor se almacena en la variable `contraseña_ingresada` pero nosotros no lo conocemos, así que no podemos saber si es el valor correcto o no.

La siguiente instrucción en el programa es el `if` con todos sus elementos. En primer lugar, Python verifica el valor de la expresión booleana para ver cuál es la siguiente instrucción que tendrá que ejecutar. En este caso la expresión es `contraseña_almacenada == contraseña_ingresada` y tiene un valor que nosotros no podemos conocer porque depende de lo que escriba el usuario cada vez que se ejecute el programa. Lo que sí sabemos es que si el valor es verdadero se ejecutará el *cuerpo del if* y que si es falso se ejecutará el *cuerpo del else*.

El *cuerpo del if* en este caso sólo tiene una instrucción que hace una asignación a la variable `mensaje` con un valor que indique una autenticación exitosa. El *cuerpo del else* también tiene sólo una asignación, pero esta vez deja un mensaje fallido. Estos dos bloques de código (el *cuerpo del if* y el *cuerpo del else*) son muy sencillos en este caso, pero podrían ser tan complicados como se quisiera: dentro de ellos puede haber casi cualquier cosa que pueda estar dentro de un programa, incluyendo asignaciones, llamados de funciones y otros condicionales [1].

Un aspecto muy importante para tener en cuenta acá es la indentación: así como cuando definimos una función tenemos que indentar el cuerpo de la función, cuando usamos instrucciones condicionales el cuerpo del if y el cuerpo del else tienen que estar consistentemente indentados.

Cuando se termine de ejecutar bien sea el cuerpo del if o el cuerpo del else, se considerará que la instrucción condicional terminó de ejecutarse y Python podrá pasar a la siguiente instrucción, que en este imprime el mensaje para el usuario.

3.4.2.1. Alternativas

Veamos ahora otras dos maneras en las que habríamos podido escribir el mismo programa.

En la primera alternativa vamos a extraer la instrucción condicional y convertirla en una función que calcule el mensaje.

```
def revisar(contraseña_almacenada: str, contraseña_ingresada: str) -> str:
    contraseña_correcta = contraseña_almacenada == contraseña_ingresada

    if contraseña_correcta:
        mensaje = "Su contraseña es correcta"
    else:
        mensaje = "La contraseña es incorrecta"
    return mensaje

contraseña_almacenada = "secreto"
contraseña_ingresada = input("Ingrese su contraseña:")
mensaje = revisar(contraseña_almacenada, contraseña_ingresada)
print(mensaje)
```

Este pequeño cambio hace mucho más evidente lo que dijimos antes: la instrucción condicional, aunque tuviera varios elementos por dentro, era fundamentalmente sólo una instrucción que calculaba el valor de la variable `mensaje`.

Esta nueva versión también tiene un cambio con respecto a la expresión condicional: la comparación no se hace ahora dentro del if sino que se hace antes y su resultado se almacena en una variable. Aunque funcionalmente es equivalente, es posible que esta versión sea un poco más legible y haga que la instrucción condicional sea más fácil de entender. En ejercicios más avanzados, en los que las condiciones son mucho más largas y complejas, es recomendable aplicar esta técnica para simplificar el código y disminuir la posibilidad de introducir errores.

La siguiente es la segunda alternativa al código original: en este caso no hay una función adicional pero tampoco hay un `else`.

```
contraseña_almacenada = "secreto"
contraseña_ingresada = input("Ingrese su contraseña:")

mensaje = "La contraseña es incorrecta"
if contraseña_almacenada == contraseña_ingresada:
    mensaje = "Su contraseña es correcta"

print(mensaje)
```

Como se ve en este programa, no es obligatorio que cada instrucción condicional tenga un bloque `else`: sólo el bloque `if` tiene que existir siempre en una instrucción condicional. En estos casos, la condición sirve para saber si un bloque de instrucciones tiene que ejecutarse o no.

Volviendo a nuestro ejemplo, el bloque condicional revisa que la contraseña sea correcta y, en caso afirmativo, le asigna a la variable `mensaje` la cadena que informa del éxito en la autenticación. Como en este caso la variable `mensaje` siempre tiene que tener un valor, bien sea exitoso o no, tuvimos que asignarle un valor inicial antes de llegar a la instrucción condicional. Si no lo hubiéramos hecho, la variable `mensaje` no existiría cuando el usuario pusiera la contraseña equivocada y nuestro programa habría fallado al momento de intentar imprimir el mensaje:

```
NameError: name 'mensaje' is not defined
```

3.4.2.2. Ejercicios

- Queremos escribir una función que nos diga quién ganará en el lanzamiento de una moneda. Escriba una función que reciba el nombre de la persona que pidió ‘cara’, el nombre de la persona que pidió ‘sello’ y el resultado del lanzamiento (‘cara’ o ‘sello’) y responda con el nombre de la persona que haya acertado.

3.4.3. Condicionales en Python: IF-ELIF-ELSE

Vimos ya que una instrucción condicional puede tener una pareja de bloques if y else, o puede tener sólo un bloque if. Ahora agregaremos la posibilidad de tener más de una condición, para lo cual tenemos que introducir los bloques que en Python se llaman `elif` y que en muchos otros lenguajes se conocen como else if.

Veamos un ejemplo de una instrucción que incluya varias condiciones. Suponga que las funciones `longitud`, `tiene_numeros`, `tiene_minusculas`, `tiene_mayusculas` y `cambiar_contrasena` ya están definidas en alguna otra parte del programa. El bloque condicional empieza con la palabra reservada `if` y una condición; de ahí en adelante, cada condición se acompaña de la palabra reservada `elif`. Al final encontramos un bloque ELSE que, como todos los bloques ELSE que hemos estudiado, incluye la palabra `else` pero no tiene una condición asociada.

```
nueva = input("Introduzca su nueva contraseña:")

if longitud(nueva) < 8:
    mensaje = "La nueva contraseña debe tener al menos 8 caracteres"

elif nueva == contrasena_anterior:
    mensaje = "La nueva contraseña no puede ser igual a la anterior"

elif nueva.isalnum():
    mensaje = "La nueva contraseña debe tener signos de puntuación"

elif not tiene_numeros(nueva):
    mensaje = "La nueva contraseña debe tener al menos un número"

elif not tiene_mayusculas(nueva):
    mensaje = "La nueva contraseña debe tener al menos una letra mayúscula"

elif not tiene_minúsculas(nueva):
    mensaje = "La nueva contraseña debe tener al menos una letra minúscula"

else:
    cambiar_contrasena(nueva)
    mensaje = "La contraseña se cambió exitosamente"

print(mensaje)
```

En este programa lo que tenemos es una sola instrucción condicional que incluye la evaluación de varias condiciones. La primera condición que se evalúa es la que tiene que ver con la longitud de la cadena (`longitud(nueva) < 8`) para ver si se ejecuta el primer cuerpo o no. Si la condición es falsa, se pasa a revisar la segunda condición. Lo interesante es si la condición es verdadera, porque entonces se ejecuta el cuerpo del IF y se pasa a la siguiente instrucción del programa. Es decir que se pasa a imprimir el mensaje.

Esto es muy importante y debe quedar muy claro: cuando se encuentra una condición verdadera, se ejecuta el cuerpo asociado a la condición y se termina la ejecución del condicional sin importar si más adelante había otras condiciones que también fueran verdaderas.

La implicación de esto para nuestro programa es que el mensaje que le vamos a mostrar al usuario es el mensaje que corresponda al primer error que hayamos encontrado. Por ejemplo, si la nueva contraseña no tenía signos de puntuación y no tenía números, el usuario sólo se enterará del primer problema.

Finalmente, analicemos las estructuras de los siguientes condicionales. En el primer condicional, usaremos bloques ELIF.

```
if cond1:
    bloque1()
elif cond2:
    bloque2()
elif cond3:
    bloque3()
else:
    bloque_else()
```

Ahora veamos una estructura equivalente pero que no utiliza bloques ELIF.

```
if cond1:
    bloque1()
if not cond1 and cond2:
    bloque2()
if not cond1 and not cond2 and cond3:
    bloque3()
if not cond1 and not cond2 and not cond3:
    bloque_else()
```

En este segundo caso se remplazaron los bloques ELIF por bloques IF y se volvieron mucho más complejas las condiciones. Por ejemplo, en la primera estructura el llamado `bloque2()` sólo se podía ejecutar si la condición `cond1` no había sido verdadera y si la condición `cond2` sí lo había sido. En la segunda estructura encontramos exactamente las mismas condiciones, pero de una manera mucho más explícita que se va complicando progresivamente.

Esto nos demuestra la ventaja que nos trae utilizar bloques ELIF para ayudar a simplificar y hacer más comprensible el código.

3.4.3.1. Condicionales consecutivos

Suponga ahora que queremos cambiar nuestro programa para que al usuario no se le informe sólo el primer problema que se encuentre con su nueva contraseña, sino que se le digan de una vez todos los problemas que tenga. En este caso la estructura basada en IF-ELIF-ELSE ya no nos sirve porque necesitamos que *siempre* se verifiquen *todas* las condiciones.

Para lograr que esto pase, lo único que podemos hacer es separar las condiciones en instrucciones diferentes, asegurando así que todas sean verificadas. Es decir que, en lugar de tener un IF seguido de varios ELIF, en los cuales sólo 1 condición puede ser cierta a la vez, vamos a tener varios IF totalmente desconectados. Observe esto en la siguiente versión de nuestro programa:

```
nueva = input("Introduzca su nueva contraseña:")
contraseña_correcta = True
mensaje = ""

if longitud(nueva) < 8:
    contraseña_correcta = False
    mensaje += "La nueva contraseña debe tener al menos 8 caracteres" + "\n"

if nueva == contraseña_anterior:
    contraseña_correcta = False
    mensaje += "La nueva contraseña no puede ser igual a la anterior" + "\n"

if nueva.isalnum():
    contraseña_correcta = False
    mensaje += "La nueva contraseña debe tener signos de puntuación" + "\n"

if not tiene_numeros(nueva):
    contraseña_correcta = False
    mensaje += "La nueva contraseña debe tener al menos un número" + "\n"

if not tiene_mayusculas(nueva):
    contraseña_correcta = False
    mensaje += "La nueva contraseña debe tener al menos una letra mayúscula" + "\n"

if not tiene_minúsculas(nueva):
    contraseña_correcta = False
    mensaje += "La nueva contraseña debe tener al menos una letra minúscula" + "\n"

if contraseña_correcta:
    cambiar_contraseña(nueva)
    mensaje = "La contraseña se cambió exitosamente"

print(mensaje)
```

Si seguimos la traza de ejecución de este programa veremos que, después de que se ejecuten las primeras tres asignaciones, Python revisará la condición del primer IF: si es verdadera se ejecutará el cuerpo y se pasará a la siguiente instrucción; si es falsa, se pasará a la siguiente instrucción sin ejecutar nada mas. En este caso, la siguiente instrucción será el segundo IF que se evaluará sin tener en cuenta el valor de la condición del primer IF. Es decir que estamos hablando de instrucciones condicionales completamente independientes la una de la otra. Este mismo proceso se repetirá para cada una de las instrucciones siguientes.

Ahora bien, sabemos que la función `cambiar_contraseña` sólo debe ejecutarse si la nueva contraseña cumple con todos los requisitos de forma que se revisaron en las condiciones anteriores. Si le asignáramos a esas condiciones los nombres P, Q, R, S, T y U, entonces el último bloque IF tendría que escribirse de la siguiente manera:

```
if not(P or Q or R or S or T or U):
    cambiar_contraseña(nueva)
    mensaje = "La contraseña se cambió exitosamente"
```

Esta condición no es muy fácil de leer porque los nombres de variables no son explícitos, pero si pusiéramos nombres más claros terminaríamos con una expresión muy larga y también difícil de leer. También podríamos haber escrito las expresiones originales, pero eso significaría que cada una la evaluaríamos dos veces: una vez en el IF que la verifique y otra vez en el IF para cambiar la contraseña.

Por estas razones, en nuestro programa decidimos incluir una nueva variable llamada `contraseña_correcta`: cuando inicia el programa suponemos que la nueva contraseña que ingresó el usuario cumple con todas las restricciones, así que inicializamos la variable en el valor `True`. Luego, en cada una de las instrucciones condicionales cambiamos el valor de la variable a falso si descubrimos que la nueva contraseña no cumple alguna de las restricciones.

Como sabemos que si la nueva contraseña no cumple con *al menos una* de las reglas entonces no debemos hacer el cambio, en el último IF basta con revisar el valor de la variable `contraseña_correcta`: si tiene valor verdadero significa que no se encontró ningún problema con la nueva contraseña; si tiene valor falso significa que se encontró *al menos un* problema la nueva contraseña.

El último aspecto para revisar de este programa es el que tiene que ver con el mensaje. Cuando se inicia la ejecución tenemos un mensaje vacío. Luego, cada vez que se encuentra que no se cumple con una restricción se concatena un nuevo mensaje al mensaje que se tenía. Esto quiere decir, por ejemplo, que si no se cumple con ninguna de las restricciones entonces la variable `mensaje` tendrá la información de todas las restricciones que no se cumplieron.

Note que en el último IF, el que cambia la contraseña si no hubo ningún problema, no se concatena nada al mensaje, sino que se reemplaza por un mensaje de éxito.

Recuerde: un `elif` depende del `if` anterior mientras que un `if` es independiente de lo que haya pasado antes.

3.4.3.2. Ejercicios

1. Modifique el programa que cambia la contraseña para que en caso de que no se pueda cambiar la contraseña se le informe al usuario no sólo las restricciones que no se cumplieron sino también las condiciones que sí se cumplieron.
2. El valor de un peaje depende del tipo de vehículo (categoría) que pase y de la cantidad de ejes que tenga: automóviles, camionetas y camperos (AUTO) deben pagar 10000; buses y busetas (BUS), 14300; camiones grandes de 2 ejes (CAMION), 16000; camiones de 3 y 4 ejes (CAMION), 28100; camiones de 5 ejes (CAMION), 37700; y camiones de 6 o más ejes (CAMION) deben pagar 41400. Adicionalmente, un vehículo podría tener un descuento especial del 15% por paso frecuente. Escriba una función que reciba el tipo de un vehículo ('AUTO', 'BUS' o 'CAMION') y si tiene descuento especial y calcule el valor que debe pagar en el peaje.

3.4.4. Aspectos metodológicos

Como hemos visto con algunos de los ejemplos anteriores, usualmente el mismo problema puede resolverse correctamente usando estructuras diferentes. A continuación exploramos cuatro formas de resolver el mismo problema: queremos saber si un número es positivo y además es menor de 10.

3.4.4.1. Condicionales anidados

El siguiente bloque muestra la definición de una función que resuelve el problema utilizando condicionales anidados:

```
def es_positivo_de_un_solo_digito_v1(x: int)->bool:
    if x > 0:
        if x < 10:
            return True
        else:
            return False
    else:
        return False
```

Llamamos condicional anidado a la situación en la cual hay una instrucción condicional dentro del cuerpo de otra instrucción condicional.

En el caso de nuestro ejemplo, lo que se hizo en este ejemplo fue analizar por partes el problema. En primer lugar se decidió que, si el número es positivo, entonces se debe revisar la segunda condición. Esta revisión quedó entonces anidada en un condicional dentro del cuerpo del primer if. Por otra parte, si el número es negativo no hay necesidad de revisar la segunda condición y se puede retornar `False` inmediatamente.

Sin embargo, este ejemplo podría simplificarse un poco si se analizan las condiciones que llevan al único caso en que se retorna `True` en esta función: para que se pueda llegar a este punto es necesario que las dos condiciones sean verdaderas simultáneamente, es decir que `x > 0` y `x < 10`. En todos los otros casos, la función debería retornar `False`.

Si llevamos eso en al código llegamos a la segunda versión de la función:

```
def es_positivo_de_un_solo_digito_v2(x: int)->bool:
    if x > 0 and x < 10:
        return True
    else:
        return False
```

No se puede decir ni que el uso de condicionales anidados sea una mala idea ni que armar expresiones condicionales más completas sea una buena idea. De hecho, siguiendo esta estrategia es muy fácil terminar con estructuras muy difíciles de entender y, por ende, de mantener, extender y corregir cuando tengan problemas.

Como siempre, mantener la claridad del código debería ser una prioridad para tomar decisiones con respecto a su estructura.

3.4.4.2. Returns

A pesar de sus diferencias, los dos ejemplos anteriores coinciden en un aspecto: tan pronto encuentran la respuesta, retornan verdadero o falso y terminan la ejecución de la función. Aunque esto no está necesariamente mal, una práctica que suele hacer el código más fácil de entender y por ende mucho más mantenable, es limitar la cantidad de puntos en los cuales se utilice la instrucción `return`. Por ejemplo, la última versión de la función puede remplazarse por la siguiente versión remplazando los `return` por asignaciones a una variable y dejando sólo un `return` al final de la función.

```
def es_positivo_de_un_solo_digito_v3(x: int)->bool:
    resultado = False
    if x > 0 and x < 10:
        resultado = True

    return resultado
```

En esta función tan pequeña no es fácil apreciar que esta nueva versión es mucho más fácil de mantener que la versión anterior. Sin embargo, nuestra recomendación es que minimice la cantidad de puntos en los cuales usted retorne un valor desde una función.

3.4.4.3. Expresiones vs. condicionales

En varios casos, incluyendo el que hemos estado trabajando, existen formas de remplazar las instrucciones condicionales por expresiones booleanas. Eso es lo que hemos hecho en la cuarta y última versión de la función: en lugar de calcular un resultado y luego retornarlo, acá el valor mismo de la expresión es el retorno de la función.

```
def es_positivo_de_un_solo_digito_v4(x: int)->bool:
    return x > 0 and x < 10
```

Esta estrategia funciona muy bien cuando las condiciones son fáciles de leer y entender. Cuando las condiciones son mucho más complicadas probablemente sea mejor tener la estructura de las instrucciones condicionales, incluso si son anidadas, para ayudar a quien lea el código a entender su propósito.

3.4.5. Caso de estudio: el mayor de 4 números

En esta sección vamos a estudiar un problema recurrente y muy importante que debe resolverse utilizando instrucciones condicionales, pero puede resolverse de maneras diferentes. Empezaremos planteándole el problema para que usted lo resuelva y luego pasaremos a discutir nuestras alternativas de solución. Esperamos que usted vaya comparando nuestras soluciones con la suya.

3.4.5.1. Enunciado del problema

Escriba una función que reciba por parámetro cuatro números enteros y devuelva el mayor de estos. Si hay dos o más iguales y mayores, retorna cualquiera de estos. La signatura de la función debe ser:

```
def mayor(a: int, b: int, c:int, d:int)->int:
```

Escriba su solución y compárela con las que nosotros proponemos a continuación.

3.4.5.2. Solución 1: múltiples returns

La primera solución es la solución que es más natural para este problema: si nos damos cuenta que el valor `a` es mayor al valor `b` y es mayor al valor `c` y es mayor al valor `d`, entonces tiene que ser que `a` es el mayor valor. Algo similar se hace con los otros parámetros.

```
def mayor(a: int, b: int, c:int, d:int)->int:
    if (a >= b) and (a >= c) and (a >= d):
        return a
    elif (b >= a) and (b >= c) and (b >= d):
        return b
    elif (c >= 1) and (c >= b) and (c >= d):
        return c
    else:
        return d
```

El problema con esta solución es que requiere mucho cuidado en su elaboración: es fácil tanto intercambiar dos variables como olvidar hacer una de las comparaciones. Esta solución además tiene como limitante que agregar un nuevo elemento para calificar requiere modificaciones sobre prácticamente toda la función.

3.4.5.3. Solución 2: un solo return

La segunda solución no dista mucho de la primera: el único cambio que se hizo fue eliminar las instrucciones `return` que estaban dentro de cada IF.

```
def mayor(a: int, b: int, c:int, d:int)->int:
    if (a >= b) and (a >= c) and (a >= d):
        mayor = a
    elif (b >= a) and (b >= c) and (b >= d):
        mayor = b
    elif (c >= 1) and (c >= b) and (c >= d):
        mayor = c
    else:
        mayor = d

    return mayor
```

3.4.5.4. Solución 3: aproximación algorítmica

La tercera solución es muy diferente de las anteriores y tiene una aproximación mucho más algorítmica, en la que los cuatro valores van analizándose uno por uno. Primero observemos con cuidado la nueva función hasta estar convencidos de que funciona.

```
def mayor(a: int, b: int, c:int, d:int)->int:
    mayor = a
    if (b > mayor):
        mayor = b
    if (c > mayor):
        mayor = c
    if (d > mayor):
        mayor = d

    return mayor
```

Al igual que en la segunda solución, tenemos una variable llamada `mayor` donde debería quedar la respuesta porque es la variable que vamos a retornar al final.

La primera instrucción de la función le asigna el valor de `a` a la variable `mayor`, implicando que `a` es el mayor valor. ¿Es esto correcto? No podemos saber si será correcto al final, pero lo que sí podemos decir con seguridad es que, si sólo hubiera un valor, entonces `a` sería el mayor. Esto es tan trivial que a veces resulta difícil al leerlo: el mayor en un conjunto que tiene sólo un elemento es ese único elemento.

¿Qué hace la primera instrucción condicional? En este caso hemos ampliado nuestro conjunto y ahora tenemos también a `b`, así que comparamos a `b` con `a` para ver cuál es el mayor. Si resulta que `b` era mayor, entonces dejamos en la variable `mayor` a `b`. Si no era mayor, entonces dejamos el valor que ya teníamos.

¿Qué hace la segunda instrucción condicional? Acá es donde todo se pone interesante porque agregamos a `c` a nuestro conjunto de valores y lo comparamos con el `mayor`. Esto quiere decir que lo comparamos con `a` o lo comparamos con `b`, pero no lo comparamos con los dos porque sería inútil: lo único que nos interesa saber es si `c`

es mayor que el valor mayor que se había encontrado hasta el momento. Note que tampoco podemos saber si el mayor era `a` o `b`. Sólo sabemos que el valor mayor está dentro de la variable `mayor`. Al finalizar este condicional dentro de la variable también podría estar el valor de `c`, pero sólo si este fuera mayor que el mayor valor entre `a` y `b`.

¿Qué hace la tercera instrucción condicional? Lo mismo que la instrucción anterior: compara a `d` con el mayor valor que se hubiera encontrado entre `a`, `b` y `c` y, si resulta que `d` es mayor, entonces remplaza el valor de `mayor` con el valor de `d`.

Al llegar a la última instrucción de la función podemos estar seguros que dentro de la variable `mayor` se encuentra el mayor valor entre `a`, `b`, `c` y `d`, a pesar de que nunca hayamos comparado los cuatro valores entre ellos.

Esta estrategia es mucho más conveniente que la de las otras versiones de la función por varias razones. Por un lado, el código es mucho más sencillo y tiene menos comparaciones: es más fácil de leer y tiene menos posibilidades de tener errores. Por otro lado, es clara cuál era la intención del programador y, si ahora hubiera un quinto valor para comparar, sería clarísimo cómo debería modificarse la función para soportar este nuevo valor.

3.4.6. Ejercicios

- Modifique el ejercicio que retorna el número mayor para que retorne el número del parámetro en el que se encuentra el número mayor. Si hay dos o más iguales y mayores, debe retornar el número de parámetro menor (ej. si el primer y el tercer parámetro eran los mayores, debe retornar el número 1).
- La calificación final de un estudiante en un curso depende de las calificaciones que obtenga en 3 exámenes, pero con unas reglas especiales. Si el estudiante sacó más de 4.0 sobre 5.00 en el tercer examen (el examen final), la nota en el curso será la nota del examen final. Si el estudiante saca menos de 2.0 en el examen final, ese examen valdrá el 50% de la nota y los otros dos exámenes valdrán el 25% cada uno. En cualquier otro caso, los exámenes pesarán lo mismo para calcular la nota final. Escriba una función que dadas las notas de los tres exámenes calcule la nota del estudiante en el curso.
- En muchos torneos de fútbol es usual que dos equipos jueguen dos partidos para definir cuál es el mejor de ellos: uno en el estadio del primer equipo y otro en el estadio del segundo equipo. También es usual que, en caso de empate en la cantidad de partidos ganados, los goles de los equipos visitantes cuenten por dos al momento de calcular la diferencia de goles. Escriba una función para calcular el ganador entre dos equipos. La función debe recibir el nombre del primer equipo (A), el nombre del segundo equipo (B), los goles que hizo A de local, los goles que hizo B de visitante, los goles que hizo A de visitante y los goles que hizo B de local. La función debe retornar el nombre del ganador de la serie o la cadena "EMPATE" si hubo un empate entre los equipos.
- Las denominaciones de las monedas actualmente disponibles en un país son: 20, 50, 100, 200, 500 y 1000. Escriba una función que reciba la cantidad de monedas de cada denominación que tiene una persona y el valor de un producto y le diga si es posible pagar el producto con el dinero en efectivo que tiene. Ayuda: tiene que revisar si tiene suficiente dinero y, si tiene más de lo necesario, si es posible que le den el cambio con las denominaciones de monedas que se encuentran en circulación.
- Picas y Fijas* es un juego en el que dos personas intentan adivinar un número de 4 dígitos que mantiene en secreto el otro jugador. En cada turno, un jugador propone un número de 4 dígitos y el otro jugador debe informar la cantidad de *picas* y la cantidad de *fijas* de ese número. Cada *pica* significa que en el número propuesto hay un dígito que también está en el número secreto, pero en una posición diferente. Cada *fija* significa que en el número propuesto hay un número que también está en el número secreto y que está en la misma posición. Por ejemplo, si el número secreto es 5678 y el número que el otro jugador propone es 6579, la respuesta sería "2 picas y 1 fija" porque 5 y 6 están en las posiciones equivocadas y porque el 7 está en la posición correcta. El ganador del juego es el jugador que encuentre el número del otro en la menor cantidad de intentos. En este ejercicio usted debe escribir dos funciones: la primera calculará la cantidad de picas dados un número secreto entre 1000 y 9999, y un número propuesto también entre 1000 y 9999; la segunda función calculará la cantidad de fijas y recibirá también un número secreto y un número propuesto.
- Un número primo es un número que es divisible sólo por 1 y por sí mismo. Los primeros 10 números primos son 2, 3, 5, 7, 11, 13, 17, 19, 23 y 29. Escriba una función que dado un número diga cuál es el menor de los primeros diez números primos por el que es divisible o retorne -1 si no es divisible por ninguno de esos números.
- En una competencia sólo pueden participar estudiantes universitarios que sean menores de 23 años o que cumplan 23 años durante el año en curso. Además, pueden participar todos los estudiantes universitarios que hayan cursado menos de 2 años de estudios en la Universidad. Escriba una función que reciba el año de nacimiento de una persona y el año de entrada a la universidad y retorne un valor de verdad indicando si la persona puede participar o no.
- En una ciudad existe una restricción de circulación para los vehículos que depende del número de la placa, del tipo de vehículo, del día de la semana y de la hora del día. Los vehículos particulares sólo tienen restricción de lunes a viernes, dependiendo del último dígito de su placa: los que terminen en un número par no podrán circular entre 6:00 y 8:30 y 15:00 y 19:30 en los días del mes que sean pares; los que terminen en un número impar no podrán circular en los mismos horarios, pero de los días del mes que sean impares. La restricción

para los taxis va desde las 5:30 hasta las 21:30, de lunes a sábado: los taxis cuya placa termine en el mismo dígito en que termine el día del mes no podrán circular ese día. Escriba una función que diga si un vehículo puede circular o no dados: el tipo de vehículo (str, TAXI o PARTICULAR), la placa (str, por ejemplo DMZ042), el día del mes (int, entre 1 y 31), el día de la semana (str - Lunes, Martes, Miércoles, Jueves, Viernes, Sábado o Domingo), la hora (int, entre 0 y 23) y el minuto (int, entre 0 y 59).

3.4.7. Más allá de Python

Además de las estructuras basadas en IF-ELIF-ELSE, muchos lenguajes ofrecen instrucciones condicionales basados en *switch* y en *operadores ternarios*. Estos últimos son de mucha utilidad porque permiten seleccionar valores basados en condiciones, pero sin tener que utilizar demasiado espacio. Por ejemplo, este código se podría usar en Java para seleccionar una cadena de caracteres basado en el valor de la variable booleana `exito`:

```
respuesta = exito ? "sí": "no";
```

Lo que haría la máquina virtual de Java (JVM) al encontrar este código sería revisar si la variable `exito` es verdadera y, en caso afirmativo, asignarle a `respuesta` el valor “sí”; en caso negativo, asignarle el valor “no”.

En Python existe una estructura similar llamada *expresión condicional*. La traducción del ejemplo anterior a Python sería:

```
respuesta = "sí" if exito else "no"
```

La interpretación sería exactamente igual que en el caso de Java: se le asignará “sí” a la variable `respuesta` sólo si `exito` es verdadero y de lo contrario se le asignará “no”. Las expresiones condicionales pueden ser de mucha utilidad cuando las condiciones son tan fáciles como en el ejemplo, pero se pueden volver difíciles de leer cuando las condiciones son más elaboradas. Sin embargo, todo lo que se haría con una expresión condicional se puede hacer con un IF.

-
- [1] En realidad, también puede haber cosas que es poco usual encontrar dentro de un condicional, como declaración de funciones y llamados para importar módulos. No es muy recomendable usar estas capacidades a menos que haya motivos muy bien justificados para hacerlo.

By Mario Sánchez
© Copyright Agosto de 2020.

[Departamento de Ingeniería de Sistemas y Computación](#) - Universidad de los Andes

3.5. Más sobre cadenas de caracteres

[Print to PDF](#)

≡ On this page

[3.5.1. Aspectos de codificación \(encoding\)](#)

[3.5.2. Caracteres de control](#)

[3.5.3. Funciones y operadores sobre cadenas de caracteres](#)

[3.5.4. Métodos sobre cadenas de caracteres](#)

[3.5.4.1. Métodos versus funciones](#)

[3.5.4.2. Inmutabilidad](#)

[3.5.4.3. Métodos de str](#)

[3.5.5. Format](#)

[3.5.6. Ejercicios](#)

[3.5.7. Más allá de Python](#)

⚠ 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.

ℹ️ Objetivo de la sección

El objetivo de esta sección es explorar un poco más el tipo `str` y descubrir otras funcionalidades que ofrece Python para manipular cadenas de caracteres. Sin embargo, esta no pretende ser una guía exhaustiva sino un abrebotas para que el lector estudie la documentación Python y descubra muchas otras funcionalidades que están disponibles.

Hace años, cuando se empezaron a construir los primeros computadores programables, el objetivo principal que tenían era permitir hacer cálculos numéricos (trayectorias de misiles, órbitas de naves espaciales, análisis de estructuras, cálculos financieros). Por este motivo, cuando se diseñaron los primeros lenguajes de programación, su diseño estuvo fuertemente influenciado por la necesidad de soportar con facilidad la expresión de cálculos numéricos cada vez más complicados. En esos lenguajes era prácticamente imposible representar una cadena de caracteres.

Hoy en día la situación ha cambiado y la representación y manipulación de información textual es igual o más importante que los cálculos numéricos. Esto ha llevado a que cualquier lenguaje de programación moderno ofrezca sólidos tipos de datos para representar cadenas de caracteres, los cuales vienen acompañados de mecanismos que permiten manipular con facilidad estas cadenas de acuerdo con los requerimientos de cada aplicación. Por ejemplo, hoy en día es muy común que las aplicaciones tengan que incluir mecanismos de internacionalización (para traducir los mensajes a otros idiomas), varios alfabetos diferentes (cirílico, griego, kanjis japoneses y hasta *emojis*), sistemas de cifrado (para garantizar confidencialidad), y sistemas de búsqueda que encuentren términos rápidamente dentro de textos cada vez más grandes.

En esta sección vamos a continuar con el estudio del tipo `str` de Python y vamos a ver algunos de los mecanismos que ofrece para manipular cadenas de forma rápida y sencilla. Sin embargo, en esta sección no termina toda la presentación de `str`: en el siguiente capítulo presentaremos los mecanismos adicionales disponibles que requieren del estudio previo de secuencias e instrucciones repetitivas.

3.5.1. Aspectos de codificación (encoding)

Si usted está leyendo este libro es muy posible que su lengua materna sea el español, que esté acostumbrado a escribir usando el alfabeto latino y que el teclado de su computador sea un teclado ‘QWERTY’^[1]. Por esto, además de necesitar los 127 caracteres básicos de ASCII, usted también necesita algunos caracteres adicionales, como ‘ñ’, ‘Ñ’ y todas las vocales con tilde en mayúsculas y minúsculas.

Si todo esto es cierto, muy posiblemente usted ha estado utilizando un conjunto de caracteres (encoding) que extiende al conjunto ASCII y que se conoce con el nombre ISO-8859-1: además de tener los 127 caracteres de ASCII, este conjunto incluye caracteres de uso frecuente en lenguajes originados principalmente en Europa occidental. Por ejemplo: á, Á, ñ, Ñ, ç, ü, ò ^[2].

Ahora bien, si usted intercambia información con personas que viven en países donde ISO-8859-1 no es de utilidad (por ejemplo, Rusia, Japón, o todo el Sudeste Asiático) tendrá que utilizar un conjunto de caracteres mucho más grande, como el ofrecido por el estándar UNICODE que describe muchos más caracteres incluyendo los cada vez más populares *emojis* 😊. Sin embargo, si usted quiere usar caracteres definidos en UNICODE, los archivos o mensajes que usted escribe tendrán que utilizar bien sea el encoding UTF-8 o el encoding UTF-16 ^[3].

Si usted puede ver los siguientes tres caracteres (un kanji, un emoji y un carácter griego) significa que su navegador o el programa que está utilizando para leer este texto es compatible con UNICODE y con UTF-8:

- Kanji: 字
- Emoji: 😊
- Carácter griego: Δ

Con esta brevísimas explicación no esperamos que usted sea ya un experto en sistemas de codificación de caracteres, pero sí que se dé cuenta que es un asunto más complicado de lo que parece a simple vista. Mientras que sus programas se limiten a usar caracteres que son normales en su idioma, probablemente no tendrá ningún problema.

Pero si quiere que sus programas sean capaces de manejar caracteres de otros alfabetos, tendrá que ponerles atención a los asuntos de codificación si no quiere perder información o ver caracteres remplazados por marcas desconocidas.

La buena noticia es que, por defecto, Python soporta cadenas UNICODE. Así que, si está teniendo problemas con caracteres extraños dentro de sus cadenas, probablemente el problema no sea de Python sino de algo más de lo que usted está haciendo.

Otra cosa con la que usted tiene que tener cuidado es la codificación que utilice para los archivos .py en los que escriba sus programas. La recomendación que le hacemos es que se asegure de que los archivos se guarden con la codificación UTF-8. En Spyder esto se logra poniendo el siguiente comentario al principio de cada archivo:

```
# -*- coding: utf-8 -*-
```

Si lo hizo correctamente, en la esquina inferior derecha del ambiente debería ver una etiqueta que diga *Encoding*: *UTF-8*. Si en lugar de esto dice *Encoding: ASCII*, entonces cualquier carácter que usted escriba en su programa y que no sea parte de ASCII (por ejemplo 'á') se convertirá en basura cuando usted cierre el archivo y lo vuelva a abrir.

Finalmente le hacemos una recomendación con respecto al uso de caracteres *no ASCII* dentro de sus programas: está bien usar estos caracteres dentro de la documentación y comentarios, pero evite usarlos en los identificadores de variables, parámetros y nombres de funciones. Por ejemplo, en lugar de llamar a una función `calcular_año` use `calcular_anho`, y en lugar de usar la variable `máximo` use la variable `maximo`. Esta no es una regla escrita en piedra que usted tenga que seguir. Es sólo un consejo para evitar que se tope con problemas sencillos, pero a veces difíciles de diagnosticar.

3.5.2. Caracteres de control

Además de los caracteres como los que se describieron en la sección anterior, una cadena de caracteres (`str`) también puede tener caracteres que son llamados caracteres de control. Estos caracteres se pueden usar como cualquier otro dentro de una cadena.

A continuación, describimos los más importantes y que usted podría necesitar.

Carácter	Significado	Ejemplo de uso	Resultado
\n	Cambio de línea	"Hola\nMundo!"	Hola Mundo!
\t	Tabulación	"Hola\tMundo"	Hola Mundo
\	Barra invertida	"C:\usuarios\invitado"	C:\usuarios\invitado
'	Comilla sencilla (2)	"It's me!"	It's me!
"	Comilla doble	"A so called "expert""	A so called "expert"

Como es posible que con un ejemplo tan breve no se aprecie la importancia de la tabulación, a continuación presentamos un ejemplo un poco más complejo. El uso del carácter de tabulación indica que se quiere dejar espacio hasta la siguiente tabulación, con el objetivo de armar columnas más o menos definidas. Una tabulación típicamente equivale al espacio de 8 caracteres.

```
print("a\tb\tc" + "\n" + "100\t200\t300")
```

El resultado de ejecutar el programa anterior es el siguiente:

a	b	c
100	200	300

3.5.3. Funciones y operadores sobre cadenas de caracteres

En la sección anterior estudiamos los operadores relacionales y vimos que pueden aplicarse a diferentes tipos de datos. En particular, vimos que los operadores `<`, `>`, `<=`, `>=` hacen comparaciones lexicográficas. De esta manera sabemos que las siguientes expresiones son todas verdaderas:

- `"AB" < "BC"`
- `"AB" < "ab"`
- `"234" < "345"`
- `"!450" < "345"`

Los otros operadores relacionales que podemos aplicar a las cadenas de caracteres son `==` y `!=`, los cuales comparan las cadenas carácter por carácter. De esta manera, las siguientes expresiones son todas falsas:

- `"ABC" == "abc"`
- `"ABC" == "ABC!"`
- `"3" == "1+2"`
- `"Hola, Mundo!" != "Hola, Mundo!"`

Dos nuevos operadores que también se pueden aplicar sobre cadenas de caracteres son `in` y `not in`. El primero permite revisar si una cadena está incluida en otra, mientras que el segundo hace exactamente lo contrario. Veamos un ejemplo de expresiones verdaderas basadas en estos operadores:

- `"Mundo" in "Hola, Mundo!"`
- `"mundo" not in "Hola, Mundo!"`
- `"!" in "Hola, Mundo!"`
- `'?' not in "Hola, Mundo!"`
- `"Hola, Mundo!" in "Hola, Mundo!"`

Como vemos, estos operadores hacen comparaciones que distinguen entre mayúsculas y minúsculas. Esto debe tenerse en cuenta para evitar tener falsos negativos y falsos positivos.

Finalmente, Python nos ofrece la función `len` que podemos utilizar para conocer el tamaño de una cadena.

- `len("Hola, Mundo!")`
- `len("Hola,\nMundo!")`
- `len("")`
- `len(" ")`
- `len("\t")`

El resultado de las primeras dos invocaciones es el mismo (12). Esto porque el carácter de control `\n` es un solo carácter aunque se represente con dos.

El resultado de la tercera invocación es 0, porque es una cadena vacía (no tiene ningún carácter).

El resultado de las últimas invocaciones es 1 porque se trata de cadenas con sólo un carácter: la cuarta tiene sólo un espacio mientras que la quinta sólo tiene una tabulación.

3.5.4. Métodos sobre cadenas de caracteres

3.5.4.1. Métodos versus funciones

Además de todo lo que ya hemos estudiado, el tipo `str` también ofrece una serie de funciones que pueden utilizarse para transformar cadenas de caracteres. Debido a la forma particular en la que estas funciones están declaradas dentro del módulo `str`, estas funciones se conocen usualmente como métodos y se pueden invocar de una forma diferente a como lo haríamos con funciones como las que ya conocíamos [4].

Para ilustrar este punto, estudiemos la función `lower` que está definida dentro del contexto del módulo `str`. En este módulo es donde se define el tipo que hemos estado usando para las cadenas de caracteres. Usando la función `help` podemos consultar la documentación de esta función:

```
>>> help(str.lower)
Help on method_descriptor:

lower(self, /)
    Return a copy of the string converted to lowercase.
```

Lo que esto nos dice es que si invocamos esta función usando como parámetro una cadena de caracteres vamos a recibir como respuesta *una copia* de la cadena en la cual las letras se habrán pasado todas a minúsculas. Veamos una prueba que nos ayuda a comprobar que esta interpretación es correcta:

```
>>> str.lower('ABcdEg')
'abcdeg'
```

Sin embargo, la documentación de la función también nos dice que esta función es un método, así que podemos invocarla de una forma un poco diferente que en la mayoría de los casos resulta más conveniente:

```
>>> 'ABcdEg'.lower()
'abcdeg'
```

Observe como el parámetro de la función pasó a ocupar el lugar frente al carácter `.`, dejando a la función sin parámetro.

La pregunta que cabe hacer en este punto es ¿cómo sabe Python que la función está definida dentro del módulo `str`? La respuesta está en que Python primero va a verificar el tipo del valor `'ABcdEg'` y como el tipo es `str` significa que las funciones que se llamen sobre este valor tienen que estar implementadas en el módulo `str`.

Hagamos ahora el mismo ejercicio con la función llamada `str.zfill`:

```
>>> help(str.zfill)
Help on method_descriptor:

zfill(self, width, /)
    Pad a numeric string with zeros on the left, to fill a field of the given width.

    The string is never truncated.
```

La documentación nos dice que esta función le va a agregar ceros a la izquierda a una cadena, hasta lograr una cadena de la longitud indicada. La documentación también nos dice que esta función requiere dos parámetros (`self` y `width`). Como es un método, esta función podría llamarse sobre una cadena de la siguiente manera:

```
>>> "abc".zfill(10)
'0000000abc'
```

Si quisieramos invocarla de la manera *normal*, tendríamos que convertir la cadena en el primer parámetro e indicar en qué módulo está definida la función. El resultado sería el siguiente:

```
>>> str.zfill("abc", 10)
'0000000abc'
```

En resumen: el módulo `str` ofrece unas funciones llamadas métodos que implementan una gran cantidad de funcionalidades interesantes para trabajar con cadenas de caracteres. Estas funciones se pueden invocar de dos formas: como funciones, usando un llamado de la forma `str.funcion(cadena, ...otros parámetros ...)`, o como métodos, usando un llamado de la forma `cadena.metodo(... otros parámetros ...)`. De acá en adelante nosotros vamos a usar estas funciones de la segunda forma (como métodos).

3.5.4.2. Inmutabilidad

Antes de presentar los métodos más utilizados de `str` es muy importante hablar de la *inmutabilidad* de las cadenas de caracteres en Python. Esto significa que cuando Python construye una cadena no puede modificarla para cambiar su contenido. Si queremos hacerle algún cambio a la cadena, lo único que Python puede hacer es construir una nueva cadena con los cambios requeridos.

Observemos un ejemplo utilizando el método `lower` que ya estudiamos y los operadores `is` (comparación de identidad) y `==` (comparación de igualdad).

```
original = "texto original"
minusculas = original.lower()
print("Son iguales:", original == minusculas)
print("Son el mismo:", original is minusculas)
```

El resultado de ejecutar este pequeño programa es el siguiente:

```
Son iguales: True
Son el mismo: False
```

Lo que esto nos muestra es que, como dice la documentación del método, `lower()` retorna una *copia* de la cadena en la cual se hayan hecho las modificaciones para que todas las letras sean minúsculas. En el caso de nuestro ejemplo no hubo ninguna modificación (la comparación de igualdad es exitosa) y de todas maneras el método creó una copia de la cadena original.

Recuerde: Las cadenas de caracteres en Python son *inmutables*. Todas las operaciones que se realicen sobre una cadena siempre van a producir una cadena nueva.

3.5.4.3. Métodos de str

Los siguientes son algunos de los métodos de `str` que podría necesitar con más frecuencia cuando trabaje con cadenas de caracteres.

3.5.4.3.1. Mayúsculas y minúsculas

Nombre	Descripción	Ejemplo	Resultado
lower	Reemplaza todas las mayúsculas por minúsculas	"Hola, Mundo!".lower()	'hola, mundo!'
upper	Reemplaza todas las minúsculas por mayúsculas	"Hola, Mundo!".upper()	'HOLA, MUNDO!'
title	Le pone mayúscula inicial a todas las palabras de la cadena	"Hola, Mundo!".title()	'Hola, Mundo!'
swapcase	Intercambia mayúsculas por minúsculas y viceversa	"Hola, Mundo!".swapcase()	'hOLA, mUNDO!'

3.5.4.3.2. Análisis de cadenas

Nombre	Descripción	Ejemplo	Resultado
find	Busca la primera posición en la que aparezca la cadena buscada. Si no la encuentra retorna -1.	"Hola, Mundo!".find('o')	1
rfind	Busca la primera posición en la que aparezca la cadena buscada, empezando por la derecha. Si no la encuentra retorna -1.	"Hola, Mundo!".rfind('o')	10
count	Cuenta cuántas veces está la cadena indicada en otra cadena. Si no la encuentra se produce un error.	"Hola, Mundo!".count('o')	2
isnumeric	Revisa si todos los caracteres de una cadena son números	"Hola, Mundo!".isnumeric()	False

3.5.4.3.3. Manipulación de cadenas

Nombre	Descripción	Ejemplo	Resultado
replace	Reemplaza algunos elementos de una cadena con otros elementos que llegan como parámetro. El reemplazo puede ser la cadena vacía para eliminar los elementos buscados.	"Hola, Mundo!".replace('o', ''67')	'Hola, Mund67!'
strip	Elimina espacios y cambios de línea al final de una cadena de caracteres. No elimina ningún elemento que no se encuentre al final de la cadena.	"Hola, Mundo! \n".strip()	'Hola, Mundo!'
ljust	Amplía la cadena hasta el ancho indicado, alinea el contenido a la izquierda y llena el espacio vacío con espacios.	"Hola, Mundo!".ljust(15)	'Hola, Mundo! '
rjust	Amplía la cadena hasta el ancho indicado, alinea el contenido a la derecha y llena el espacio vacío con espacios	"Hola, Mundo!".rjust(15)	' Hola, Mundo!'
center	Amplía la cadena hasta el ancho indicado y centra el contenido.	"Hola, Mundo!".center(15)	' Hola, Mundo! '
zfill	Amplía la cadena hasta el ancho indicado, alinea el contenido a la derecha y llena el espacio vacío con caracteres <code>0</code> .	"Hola, Mundo!".zfill(15)	'000Hola, Mundo!'

3.5.4.3.4. El resto de métodos

El módulo `str` ofrece una gran cantidad de métodos adicionales que no vamos a cubrir acá pero que sí están bien descritos en la documentación de Python: <https://docs.python.org/3.6/library/stdtypes.html#string-methods>

Le recomendamos estudiar esa documentación y practicar con todos estos métodos para aprender a utilizarlos y agregarlos a su caja de herramientas de programación.

3.5.5. Format

Las últimas funcionalidades de las cadenas de caracteres que vamos a estudiar tienen que ver con el método `format`.

Este método es muy poderoso y permite utilizar repetidamente una cadena como plantilla para luego remplazar algunos fragmentos por valor particulares.

Supongamos que tenemos que hacer un programa que sea capaz de generar las siguientes cadenas:

```
Un Lamborghini Aventador del 2016 con motor de 6.5 litros es capaz de pasar de 0 a 100 kph en 2.80 segundos
Un Ferrari Enzo del 2002 con motor de 6.0 litros es capaz de pasar de 0 a 100 kph en 3.60 segundos
Un Bugatti Veyron 16.4 del 2010 con motor de 8.0 litros es capaz de pasar de 0 a 100 kph en 2.50 segundos
Un Porsche 911 GT3 del 2011 con motor de 4.0 litros es capaz de pasar de 0 a 100 kph en 3.80 segundos
Un McLaren P1 del 2013 con motor de 3.8 litros es capaz de pasar de 0 a 100 kph en 2.80 segundos
Un Pagani Huayra BC del 2017 con motor de 6.0 litros es capaz de pasar de 0 a 100 kph en 3.30 segundos
```

Todas estas cadenas tienen la misma estructura que incluye la marca del carro, el nombre del modelo, el año de lanzamiento, el tamaño del motor en litros usando una cifra decimal y el tiempo que necesita para pasar de 0 a 100 kilómetros por hora expresado en segundos con dos cifras decimales.

Estas cadenas se podrían haber armado usando concatenaciones. Sin embargo, el método `format` ofrece características para estructurar las cadenas que son muy difíciles de replicar usando sólo concatenaciones.

Tomemos como ejemplo la plantilla utilizada para generar los mensajes anteriores:

```
plantilla = "Un {0} {1} del {2:d} con motor de {3:.1f} litros es capaz de pasar de 0 a 100 kph en {4:.2f} segundos"
```

En primer lugar, la plantilla es una cadena de caracteres que tiene marcas rodeadas por los caracteres `{` y `}` en los campos donde se tendrán que insertar valores. Aunque no es obligatorio, es recomendable numerar esos campos empezando desde el 0 y llegando, en este caso, hasta el 4.

Además del número, cada campo también puede tener una especificación de su formato, la cual viene a continuación del carácter `:`. En nuestro caso, los campos 0 y 1 no tienen ningún formato, así que lo que metamos en esas posiciones se remplazará sin ningún cambio. Para el campo 2 sólo usamos el carácter `d` para indicar que se trataba de un número entero. Para los campos 3 y 4 especificamos que se trataban de números decimales y además dijimos que nos interesaba tener 1 dígito decimal en el primer caso y 2 dígitos decimales en el segundo. Esto quedó especificado usando los formatos `.1f` y `.2f` respectivamente.

Para usar una plantilla se tiene que invocar el método `format` utilizando como parámetros los valores que queremos insertar en los campos. En nuestro caso, invocamos la función `format` con 5 parámetros:

```
mensaje = plantilla.format(marca, modelo, anho, cc, sec)
```

Acá puede verse la importancia de numerar los campos: el valor marca quedará en el campo marcado con el 0, el valor modelo en el campo marcado con el 1 y así sucesivamente. Esto significa que el orden en el que aparecen los campos en la plantilla no necesariamente es el mismo orden en que se deben pasar los argumentos. También significa que, si hubiéramos marcado dos campos con el mismo número, el valor correspondiente habría aparecido dos veces en el mensaje.

Estudiemos ahora otros mensajes que tienen un poco más de complicaciones que no son fáciles de solucionar sólo usando concatenación de cadenas:

```
100 metros      5.3 s @ 126 kph
500 metros     13.6 s @ 211 kph
1000 metros    22.0 s @ 249 kph
```

Estos nuevos mensajes tienen una característica que no tenían los mensajes anteriores: los campos no están completamente llenos y están alineados a la derecha (columnas 2 y 3) o a la izquierda (columna 1). Analicemos entonces la plantilla que se utilizó en este caso

```
plantilla2 = "{0:<14}{1:>4.1f} s @ {2:>3d} kph"
```

El formato para el primer campo dice `<14`, lo cual se interpreta como que se debe alinear el contenido a la izquierda y que debe tener 14 caracteres de ancho en total. Si el contenido del campo es más corto que esto, se agregarán tantos espacios como haga falta a la derecha del contenido.

El formato para el segundo campo dice `>4.1f`. Como ya sabemos, `.1f` significa que se tratará de un número con exactamente un decimal. La otra parte, `>4` indica que el número debe alinearse a la derecha y que debe ocupar, en total, el espacio de 4 caracteres. En este caso como el número está alineado a la derecha los caracteres faltantes se agregan a la izquierda del contenido.

El formato para el tercer campo es muy similar: se espera un número entero que terminará alineado a la derecha en una columna de 3 caracteres de ancho.

Finalmente, veamos cómo se generaron las 3 cadenas invocando el método `format` sobre la plantilla:

```
plantilla2.format('100 metros', 5.3, 126)
plantilla2.format('500 metros', 13.6, 211)
plantilla2.format('1000 metros', 22, 249)
```

En resumen, cuando se construye una plantilla para usar con el método `format` se debe:

1. Marcar los campos que se van a remplazar, numerándolos de cero en adelante.

2. Para cada campo que se quiera alinear, indicar si la alineación debe ser a la izquierda (<), derecha (>) o el centro (^) y el ancho que debe tener el campo. También se puede especificar el carácter que se debe utilizar para los espacios vacíos (ver documentación).
3. Especificar el formato, especialmente si se trata de números decimales.

Se puede conseguir mucha más información sobre la especificación de plantillas usando el comando `help('FORMATTING')`. La documentación completa también está disponible en <https://docs.python.org/3.7/library/string.html#format-string-syntax>

3.5.6. Ejercicios

1. Escriba una función que reciba una cadena de caracteres y una letra. Su función debe retornar la misma cadena que recibió, pero cambiando todas las vocales por la letra que también llegó por parámetro. Por ejemplo, si la cadena original era "Hola, Mundo!" y la letra entregada fue 'l', el resultado debería ser "Hlll, Mlldl!".
2. Escriba una función que reciba dos cadenas de caracteres. La función debe retornar 1 si las cadenas son idénticas, 2 si las cadenas sólo se diferencian por las mayúsculas y minúsculas, o 0 de lo contrario.
3. Escriba una función que reciba dos cadenas de caracteres que sólo van a contener letras mayúsculas y minúsculas. La función debe retornar -1 si en un diccionario la primera cadena debería ir antes que la segunda, debe retornar 1 si la segunda cadena debe ir antes que la primera, o 0 si las dos cadenas son la misma (ignorando mayúsculas y minúsculas).
4. Escriba una función que reciba una cadena de caracteres y cuente las palabras que aparecen en la cadena. Usted puede suponer que la cadena tendrá letras (mayúsculas y minúsculas) y espacios, pero no tendrá ningún signo de puntuación ni espacios seguidos.
5. Tres equipos de fútbol participaron en un pequeño torneo en que jugaron entre ellos 3 partidos. Escriba una función que reciba el nombre de los tres equipos y los marcadores de los tres partidos, y que retorne una tabla con las posiciones de los equipos al finalizar el torneo. La función recibirá entonces 9 parámetros: primero los nombres de los tres equipos y luego 6 enteros con los marcadores de los 3 partidos. Cada partido ganado entregaba 3 puntos y cada partido empatado entregaba 1 punto. La tabla con el resultado del torneo tiene que ser una cadena de caracteres con la información organizada en columnas bien alineadas. Las columnas deben estar organizadas de la siguiente forma:
 - posición,
 - nombre del equipo,
 - puntos obtenidos,
 - partidos jugados,
 - partidos ganados,
 - partidos empatados,
 - partidos perdidos,
 - goles a favor,
 - goles en contra,
 - diferencia de goles

1. Escriba una función que, dada la altura de un edificio, retorne una cadena como en el siguiente ejemplo: "*Un objeto que cae de un edificio de 30 metros tarda 2.47 segundos en llegar al piso y alcanza una velocidad de 24.25 metros por segundo.*" Su programa debe usar las funciones de formato de cadenas.

Ayuda: El tiempo que tarda la caída es igual a la raíz cuadrada de dos veces la altura sobre la aceleración de la gravedad (9.8 m/s^2). La velocidad que alcanza el objeto es igual al tiempo de la caída multiplicado por la aceleración de la gravedad.

1. Escriba una función que dados el nombre de un país, la cantidad de habitantes en millones y el Producto Interno Bruto en millones de USD, retorne una cadena como en el siguiente ejemplo:
 "Colombia.....=45 millones= 336599 USD Million". La primera columna debe estar alineada a la izquierda, debe tener 25 caracteres y ocupar los espacios vacíos con .; la segunda columna debe estar centrada, tener 25 caracteres y ocupar los espacios vacíos con *; la tercera columna debe estar alineada a la derecha, tener 10 caracteres más el espacio ocupado por USD Million y debe ocupar los espacios vacíos con espacios.

3.5.7. Más allá de Python

Las cadenas de caracteres son de muchísima importancia en prácticamente todos los lenguajes de programación, pero hay mucha variedad en los mecanismos para representarlos y manipularlos. Por ejemplo, en lenguajes como C y C++ el uso de cadenas de caracteres no es tan sencillo como en lenguajes más modernos puesto que se construyen sobre tipos más sencillos y requieren de particular cuidado para evitar problemas en la manipulación de la memoria.

En lenguajes como Java y Python, estas limitaciones ya no existen y las cadenas se pueden usar con mucha más facilidad, aunque sigue siendo necesario poner mucha atención a ciertos detalles. Por ejemplo, en Java las cadenas de caracteres son objetos que aparentemente son como cualquier objeto otro, pero reciben un tratamiento *preferencial*: hay expresiones en el lenguaje específicas para trabajar con cadenas y el compilador procesa de forma diferente los objetos de tipo String que se construyan explícitamente dentro de un programa.

Otro aspecto común en varios lenguajes es el que tiene que ver con la inmutabilidad de las cadenas de caracteres. Así como en Python, en Java y en otros lenguajes las cadenas de caracteres son inmutables: una vez se construyen no pueden cambiar su contenido. Esto se hace por motivos de eficiencia tanto en los cálculos como en el uso de memoria, pero podría llevar a errores si no se tiene en cuenta esta condición.

En Python las cadenas de caracteres tienen otra propiedad muy interesante y es que son *secuencias*. Es decir que pueden manipularse igual que como se manipularían listas, diccionarios y otras estructuras de datos básicas del lenguaje. Este aspecto de las cadenas de caracteres se estudiará en el siguiente capítulo.

-
- [1] QUERTY hace referencia a las primeras 5 letras de la fila de letras superior del teclado.
 - [2] Si usted no puede ver alguno de estos caracteres, significa que el encoding ISO-8859-1 no está soportado o no está activo en su navegador, en el visor que esté utilizando para leer el texto, o en su computador.
 - [3] Esta última parte puede ser un poco complicada de entender, pero no es crítico que entienda todos los detalles: UNICODE define cuáles son los caracteres, cómo deberían verse y qué número le corresponde a cada uno. UTF-8 define cómo se deben representar los caracteres UNICODE usando números binarios de 8, 16, 24 o 32 bits. UTF-16 define cómo representar esos caracteres usando 16 o 32 bits.
 - [4] Siendo más precisos, `str` es en Python una *clase* y las clases definen un espacio de nombres dentro de los que están definidas un conjunto de funciones que son llamados métodos. Como se anunció en la introducción, en este libro no se tratará el tema de la programación orientada a objetos, así que no ahondaremos en el tema de la definición de clases y métodos. Sin embargo, para construir efectivamente programas Python, es necesario saber que existen librerías básicas basadas en clases, como el caso de `str`, y es necesario ser capaz de utilizar los métodos implementados en esas librerías.

By Mario Sánchez
© Copyright Agosto de 2020.

[Departamento de Ingeniería de Sistemas y Computación](#) - Universidad de los Andes

3.6. Módulos de la librería estándar

[Print to PDF](#)

On this page

[3.6.1. El módulo math](#)

[3.6.1.1. Funciones](#)

[3.6.1.2. Constantes](#)

[3.6.2. El módulo random](#)

[3.6.2.1. Valores continuos](#)

[3.6.2.2. Valores discretos](#)

[3.6.2.3. Variables aleatorias](#)

[3.6.3. Ejercicios](#)

[3.6.4. Más allá de Python](#)

⚠ 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.

ℹ️ Objetivo de la sección

El objetivo de esta sección es presentar un par de módulos para ilustrar el poder de las funcionalidades que ya están disponibles en la librería estándar de Python.

Afortunadamente vivimos en una época en la cual no tenemos que preocuparnos por programar absolutamente todo cada vez que queremos construir un nuevo programa. Por ejemplo, cuando en 1992 la compañía idSoftware empezó a trabajar en Wolfenstein 3D, tuvieron que construir desde cero todas las funcionalidades para representar espacios tridimensionales en la pantalla. Hoy en día, no sólo hay infinidad de librerías que hacen las mismas funcionalidades, sino que algunas de esas funcionalidades están implementadas dentro de las tarjetas de video para que su uso sea aún más fácil y su ejecución sea más rápida.

Por otra parte, esta realidad hace que sea cada vez más importante tener la curiosidad de buscar librerías que puedan ayudarnos con nuestro trabajo y la habilidad para aprender a usarlas rápidamente.

En esta sección enfrentamos esta realidad introduciendo dos módulos que hacen parte de la librería estándar de Python. Es decir, estos dos módulos hacen parte de la librería que debería acompañar a cualquier distribución de Python y que siempre deberíamos tener disponible. Esta librería incluye más de un centenar de módulos que cubren aspectos como procesamiento de texto, manipulación de fechas y calendarios, compresión de archivos, criptografía, comunicación por Internet e interacción con el sistema operativo.

3.6.1. El módulo math

El primer módulo que vamos a introducir es el módulo `math`, cuya documentación actual se puede encontrar en <https://docs.python.org/3.7/library/math.html>.

El módulo `math` define funciones que permiten hacer con facilidad importantes operaciones matemáticas y definen también unas constantes de uso frecuente.

La manera de importar todo lo que ofrece este módulo es a través de la instrucción `import math`.

3.6.1.1. Funciones

Las siguientes son algunas de las funciones que define el módulo y cuyo conocimiento podría ser de mucha utilidad.

- `gcd(x, y)`: función para calcular el máximo común divisor de dos números (Greatest Common Denominator).
- `log(x, base)`: función para calcular el logaritmo de un número con respecto a una base.
- `log2(x)`: función para calcular el logaritmo de un número en base 2.
- `sqrt`: función para calcular la raíz cuadrada de un número.
- `sin, cos, tan`: funciones para calcular el seno, coseno y tangente de un ángulo medido en radianes.
- `degrees`: función para convertir un ángulo en radianes a un ángulo medido en grados.
- `radians`: función para convertir un ángulo medido en grados a un ángulo medido en radianes.

3.6.1.2. Constantes

Este módulo también define unas constantes que son de utilidad tanto para hacer otros cálculos como para detectar problemas con cálculos previos. Estos valores son:

- π
- e (número de Euler)
- \inf , el valor que utiliza Python para representar el infinito.
- nan , el valor que utiliza Python para representar un número indefinido (NaN significa Not A Number).

Tenga cuidado: aunque en la mayoría de lenguajes las constantes se suelen expresar con mayúsculas, en el módulo `math` las constantes tienen nombres en minúsculas, como se ve en el siguiente fragmento:

```
>>> print("El valor de pi:", math.pi)
El valor de pi: 3.141592653589793
>>> print("El valor de e:", math.e)
El valor de e: 2.718281828459045
>>> print("El valor de infinito:", math.inf)
El valor de infinito: inf
>>> print("El valor de un número indefinido:", math.nan)
El valor de un número indefinido: nan
```

3.6.2. El módulo random

El segundo módulo que vamos a introducir es el módulo `random`, cuya documentación actual se puede encontrar en <https://docs.python.org/3.7/library/random.html>.

El módulo `random` define funciones que generan números aleatorios de acuerdo con diferentes reglas. Por ejemplo, este módulo ofrece mecanismos para generar valores continuos, valores discretos y también valores que se ajusten a las principales distribuciones aleatorias.

La manera de importar todo lo que ofrece este módulo es a través de la instrucción `import random`.

3.6.2.1. Valores continuos

Una variable aleatoria continua es una variable que puede asumir cualquiera de los valores dentro de un rango determinado, con una probabilidad que depende de la distribución asociada a la variable.

Dentro del módulo `random`, la función también llamada `random` es tal vez la más utilizada porque permite generar valores uniformemente distribuidos entre 0 y 1. Es decir, cada vez que se invoque la función `random.random()` se obtendrá un número entre 0 y 1, escogido de forma completamente aleatoria. Note que 0 es un valor posible, pero 1 está por fuera del intervalo considerado.

La gran ventaja que tiene esta distribución es que multiplicar el resultado de la función por un valor 'x' hace que se encuentren valores uniformemente distribuidos entre 0 y 'x'.

Por ejemplo, si quisieramos generar un valor aleatorio entre 0 y 7, podríamos ejecutar el siguiente código:

```
import random

valor = random.random() * 7
```

Un efecto similar se puede lograr usando la función `uniform` que recibe dos parámetros 'a' y 'b' y genera un número aleatorio en el intervalo [a, b). Note que llamar `random.uniform(a, b)` es equivalente a invocar `random.random()*(b-a) + a`.

3.6.2.2. Valores discretos

Una variable aleatoria discreta toma sólo valores discretos dentro de un rango determinado, con una probabilidad que depende de la distribución asociada a la variable. A diferencia de las variables continuas, cuando las variables son discretas los posibles valores que pueden asumir son enumerables.

La principal función para generar variables aleatorias discretas se llama `randint` y sirve para generar valores enteros entre dos números 'a' y 'b'. En el siguiente programa se usa esta función para simular el lanzamiento de un dado:

```
import random

lanzamiento = random.randint(1, 6)
```

Una función relacionada es `randrange`, que genera valores enteros desde un número inicial (start), hasta un número final (stop), con un cierto intervalo (step). Por ejemplo, si queremos un número múltiplo de 3 entre 6 y 30 podemos usar la siguiente invocación:

```
numero = random.randrange(6, 30, 3)
```

Note que esta función puede generar el número 'start' pero nunca generará el valor 'stop'.

3.6.2.3. Variables aleatorias

Finalmente, el módulo random incluye funciones para generar valores siguiendo la distribución triangular, Beta, exponencial, Gamma, Normal, y Pareto, entre otras.

Como esta no pretende ser una revisión exhaustiva sólo revisaremos la función `random.normalvariate`, que genera números distribuidos de acuerdo a una distribución normal.

```
>>> help(random.normalvariate)
Help on method normalvariate in module random:

normalvariate(mu, sigma) method of random.Random instance
    Normal distribution.

    mu is the mean, and sigma is the standard deviation.
```

Esta función requiere de un parámetro `mu` (el valor promedio de los valores en la distribución) y de un parámetro `sigma` (la desviación estándar de los valores) para generar valores que se distribuyan de forma normal de acuerdo con los parámetros. Los siguientes son 10 valores generados con esta función usando una media de 10 y una desviación estándar de 1.5:

```
# random.normalvariate(10,1.5)
0 - 9.74230603318132
1 - 9.765339949262536
2 - 10.309760658154236
3 - 10.00652736167399
4 - 8.828709896119436
5 - 9.105408757081975
6 - 8.28137647679426
7 - 9.898607684096598
8 - 7.545894557163404
9 - 10.83177690308728
```

3.6.3. Ejercicios

1. Usando la función `random.normalvariate` genere 15 números aleatorios con media 3.8 y desviación estándar de 1. Calcule ahora usted la media de los números generados y la desviación estándar. ¿Qué tan lejos están de la media y la desviación planeada? Ejecute su programa y observe cómo cambian los resultados con cada ejecución.

3.6.4. Más allá de Python

Así como Python define una librería estándar con módulos para las tareas más comunes (¡y muchas tareas no tan comunes!), la mayoría de lenguajes de programación ofrecen su propia librería estándar que debería estar disponible para todos los que usen el lenguaje. A juzgar por las discusiones y procesos legales de los últimos años sobre la propiedad y la disponibilidad de las librerías estándar de Java, se podría creer que las librerías son incluso más importantes que el lenguaje mismo.

Muchas veces, aprender a utilizar efectivamente las funcionalidades disponibles en las librerías estándar es más difícil y toma más tiempo que aprender a usar la sintaxis misma de un lenguaje. Por ejemplo, un programador experimentado debería ser capaz de dominar en un día la sintaxis del lenguaje de programación SmallTalk^[1], pero seguramente le tomaría mucho más tiempo dominar las librerías estándar que son absolutamente imprescindibles para construir incluso programas sencillos con el lenguaje. Algo similar le pasa a los programadores que pasan de Java a C# y viceversa: hay muchísimos conceptos comunes pero la principal dificultad en el proceso de aprendizaje es aprender a utilizar las librerías principales.

^[1] SmallTalk es un lenguaje de programación muy poderoso, pero con una sintaxis extremadamente pequeña. Por ejemplo, en el lenguaje mismo no existen instrucciones condicionales, sino que estas están implementadas en las librerías estándar.

3.7. Diccionarios

[Print to PDF](#)

⚠ 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.

💡 Objetivo de la sección

El objetivo de esta sección es introducir el concepto de diccionario, que en Python se implementa con el tipo de dato `dict`, y mostrar cómo puede ser de utilidad para construir programas mucho más complejos.

En las secciones anteriores hemos trabajado exclusivamente con los tipos de datos básicos de Python (`int`, `bool`, `str`, `float`). Aunque estos tipos son suficientes para muchas cosas, tienen también un problema grave: si los problemas son medianamente complejos se necesitan muchas variables y parámetros. Esto nos obliga a tener mucho cuidado para no olvidar ni confundir variables. También hace necesario ser creativo y organizado con los nombres que utilizemos.

Por ejemplo, si quisieramos hacer una función que compare tres celulares con base en la velocidad del procesador, la cantidad de memoria, la calidad de la cámara, la tecnología de la pantalla, el tamaño de la pantalla, la capacidad de la pila, el sistema operativo y su versión, necesitaríamos 27 parámetros!^[1] Además de que sería *incómodo* declarar e implementar esta función, también sería incómodo^[2] invocarla: sería muy fácil olvidar un parámetro y más aún intercambiar el orden de dos parámetros, llevando a errores difíciles de diagnosticar.

En esta sección vamos a introducir un nuevo concepto que nos permitirá simplificar un poco el problema anterior: en lugar de requerir 27 parámetros usaremos sólo 3. Este concepto, que en Python se llama diccionario y que en otros lenguajes se conoce como mapa, es parte básica del lenguaje y, como no requiere de librerías adicionales, se usa de manera extremadamente frecuente.

3.7.1. El concepto de diccionario

Un diccionario es una estructura de datos que contiene muchos valores, identificados cada uno con una llave que es única.

El término estructura de datos hace referencia a una forma de organizar datos para poder almacenarlos, modificarlos y consultarlos. Técnicamente, las variables que hemos venido utilizando son estructuras de datos pero, como sólo tienen un valor, raramente se les aplica el término a estas.

Un diccionario es entonces una estructura de datos (una forma de organizar datos), donde a cada valor que queramos almacenar se le asigna una llave (una llave es también un valor, pero además es único dentro del diccionario). Un buen ejemplo de esto es el diccionario de un lenguaje como el Español: a cada palabra (una llave) le corresponde una definición (un valor). Otro ejemplo es una red social como Twitter: a cada nombre de usuario (una llave) le corresponde un usuario (un valor) con toda su información.

Hay básicamente dos usos que nosotros le damos a un diccionario en la vida diaria. El primero es para buscar la definición de una palabra: si nosotros conocemos la palabra (la llave), podemos obtener la definición asociada a esta (el valor). El segundo uso de un diccionario es descubrir si una palabra existe o no: si no encontramos una palabra entre las llaves del diccionario, significa que la palabra no existe y que no tiene una definición. Note que un diccionario está construido pensando en que el criterio de búsqueda es la palabra y no la definición. Sería muy extraño que alguien intentara buscar la palabra en Español que corresponde a la definición “*Clase o condición a la cual está sujeta la vida de cada uno*”.

En las siguientes secciones mostraremos cómo usar este concepto en Python y mostraremos también cómo podemos construir y modificar nuestros propios diccionarios.

3.7.2. El tipo `dict` en Python

En Python los diccionarios son un elemento básico del lenguaje que, así como las cadenas, está perfectamente integrado dentro de la sintaxis del lenguaje. Para crear un diccionario basta con indicar que se quiere crear un diccionario (usando los caracteres `{}`) y separar las parejas `llave:valor` usando comas. Tomemos como ejemplo el siguiente fragmento, en el cual se han incluido cambios de línea para facilitar la lectura:

☰ On this page

[3.7.1. El concepto de diccionario](#)

[3.7.2. El tipo `dict` en Python](#)

[3.7.2.1. Nombres y valores de variables vs. nombres de llave](#)

[3.7.2.2. Tipos de llaves y valores](#)

[3.7.3. El método `get`](#)

[3.7.3.1. El valor `None`](#)

[3.7.3.2. El método `get` y el valor `None`](#)

[3.7.4. Modificación de diccionarios](#)

[3.7.4.1. Agregar una definición: primera versión](#)

[3.7.4.2. Agregar una definición: segunda versión](#)

[3.7.4.3. Agregar una definición: tercera versión](#)

[3.7.4.4. Eliminar una definición](#)

[3.7.4.5. Eliminar todas las definiciones](#)

[3.7.5. Usos de los diccionarios](#)

[3.7.5.1. Histogramas basados en diccionarios](#)

[3.7.5.2. Diccionarios como conjuntos](#)

[3.7.5.3. Diccionarios como estructuras](#)

[3.7.5.4. Diccionarios de diccionarios](#)

[3.7.6. Mutabilidad en diccionarios](#)

[3.7.6.1. Copias de diccionarios](#)

[3.7.7. Ejercicios](#)

[3.7.8. Más allá de Python](#)

```
palabras = { 'imagen' : 'Figura, representación, semejanza y apariencia de algo',
             'figura' : 'Forma exterior de alguien o de algo',
             'baraja' : 'Conjunto completo de cartas empleado para juegos de azar',
             'posibilidad' : 'Aptitud, potencia u ocasión para ser o existir algo' }
```

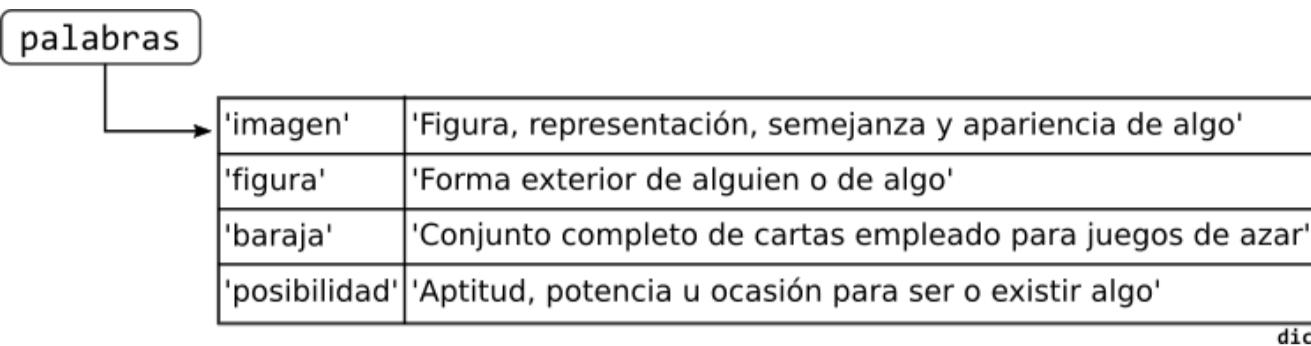


Fig. 3.2 Representación gráfica del diccionario palabras

Este fragmento crea una nueva variable con el nombre `palabras`. A diferencia de variables más sencillas que sólo tienen un valor, esta nueva variable es un diccionario y contiene 4 parejas llave-valor. También habríamos podido crear un diccionario vacío con la expresión `palabras = {}`. Observe lo que pasa si aplicamos las funciones `type()` y `len()` a nuestra nueva variable:

```
>>> type(palabras)
<class 'dict'>
>>> len(palabras)
4
```

Con `type()`, vemos que Python utiliza el término `dict` para referirse al tipo de datos de los diccionarios. Con `len()`, vemos que Python cuenta la cantidad de parejas que hay en el diccionario.

Veamos ahora cómo hacemos para consultar el contenido del diccionario:

```
>>> definicion_imagen = palabras['imagen']
>>> print(definicion_imagen)
Figura, representación, semejanza y apariencia de algo
```

En este fragmento estamos consultando el valor asociado a la cadena `'imagen'` dentro del diccionario `palabras`. Para esto usamos el nombre de la variable seguido del nombre de la llave entre paréntesis cuadrados. Note que el nombre de la llave tiene que ser idéntico al que usamos cuando creamos el diccionario. No funcionaría si usáramos `'Imagen'` o `'IMAGEN'`). Si intentamos extraer un valor usando una llave que no existe, el resultado es un error:

```
>>> definicion = palabras['IMAGEN']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'IMAGEN'
```

Para evitar este problema es posible utilizar el operador `in` que permite consultar si una llave hace parte de un diccionario o no.

```
llave = 'IMAGEN'
# Preguntar si la llave está en el diccionario antes de consultar
if llave in palabras:
    definicion = palabras[llave]
else:
    definicion = "La palabra '" + llave + "' no se encuentra en el diccionario"
```

A diferencia del ejemplo pasado, en este no se va a producir un error porque sólo consultamos el valor de la llave si estamos seguros de que se encuentra dentro del diccionario.

3.7.2.1. Nombres y valores de variables vs. nombres de llave

En la sección mostramos el uso básico de los diccionarios y la forma en la que se extrae un valor. En esta sección vamos a explorar uno de los errores más comunes que ocurren al utilizar diccionarios: confundir el nombre de una variable, con el valor de una variable y con el nombre de una llave.

En primer lugar, estudie con atención el siguiente fragmento de código y escriba los valores que debería imprimir. Si se produce algún error en alguna parte, explíquelo.

```

1 diccionario = {"llave": "valor", "palabra": "definición"}
2 llave = "llave"
3 print("1.", diccionario["llave"])
4 print("2.", diccionario[llave])
5 llave = "palabra"
6 print("3.", diccionario[llave])
7 print("4.", diccionario["palabra"])
8 print("5.", diccionario[palabra])

```

Gráficamente, el diccionario que se construye con el código anterior se ve como en la siguiente figura.

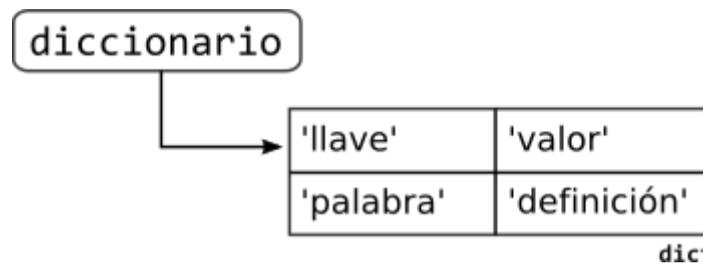


Fig. 3.3 Representación gráfica del diccionario del ejemplo

A continuación explicamos qué imprime cada llamado a la función `print` pero lo invitamos a intentar resolverlo usted antes de mirar la solución.

3.7.2.1.1. 1. diccionario["llave"]

En el primer caso, se imprime en la consola lo siguiente: `1 valor`.

Esto no debería ser una sorpresa porque se está utilizando explícitamente el nombre de la llave que está almacenada en el diccionario.

3.7.2.1.2. 2. diccionario[llave]

En el segundo caso, se imprime en la consola lo siguiente: `2 valor`.

En este caso, se está usando la variable `llave` para indicar cuál es la llave que nos interesa en el diccionario. El punto importante acá es que no nos interesa el nombre de la variable. Lo que es importante es el valor que tiene la variable. En este caso, el valor de la variable es la cadena '`llave`' y por eso el valor que se imprime es la cadena '`valor`'.

3.7.2.1.3. 3. diccionario[llave] (segunda parte)

En el tercer caso, se imprime en la consola lo siguiente: `3 definición`.

Este caso refuerza lo que dijimos en el punto anterior: aunque la variable se llama `llave`, el nombre no es importante sino el valor almacenado en ella. En este caso, la variable almacena la cadena '`palabra`', así que lo que sacamos del diccionario es el valor asociado a la llave '`palabra`'.

3.7.2.1.4. 4. diccionario["palabra"]

En el cuarto caso, se imprime en la consola lo siguiente: `4 definición`.

Esta es la versión equivalente del primer caso: usamos una cadena que es idéntica a una cadena que se encuentra dentro del diccionario.

3.7.2.1.5. 5. diccionario[palabra]

En el quinto caso, no se imprime nada en la consola porque se produce un error.

Fíjese que hasta el momento no hemos definido ninguna variable con el nombre `palabra`, así que cuando se intenta consultar el valor de esta variable, se produce un error.

3.7.2.2. Tipos de llaves y valores

En Python, las llaves y valores en un diccionario pueden ser prácticamente de cualquier tipo. Podemos tener diccionarios cuyas llaves y valores son cadenas, como en el caso de nuestro ejemplo de las palabras en Español. Podemos tener también diccionarios cuyas llaves son cadenas y sus valores son números, como en el caso de las notas que obtuvieron los estudiantes de algún curso (los nombres son las llaves y las notas son los valores). Aunque es menos usual, las llaves de un diccionario también pueden ser números, como en el caso de un diccionario que represente un edificio (las llaves son los números de los apartamentos y los valores son los nombres de quienes viven en esos apartamentos). Finalmente, los tipos de los valores pueden combinarse: más adelante exploraremos ejemplos donde las llaves son cadenas y los valores son de varios tipos diferentes.

💡 Tip

Tipos de llaves Aunque no está prohibido en el lenguaje, es recomendable evitar tener llaves de diferentes tipos en el mismo diccionario (algunas numéricas, otras cadenas, otras booleanas, etc.).

3.7.3. El método get

Como ya vimos, cuando se intenta consultar un diccionario usando una llave que no existe se produce un error. Una forma de evitar esto es consultar primero si la llave existe usando el operador `in`. Otra alternativa, muy usada porque reduce el tamaño del código, es usar el *método*[3] `get`. Este método, que se aplica sobre un diccionario, recibe como parámetros una llave y el valor que se debería retornar si la llave no existe en el diccionario.

```
llave = 'IMAGEN'
deficion = palabras.get(llave, "La palabra '" + llave + "' no se encuentra en el diccionario")
```

El fragmento anterior es equivalente al que usamos en la sección pasada: si la palabra existe en el diccionario, en la variable `deficion` queda el valor contenido en el diccionario; de lo contrario, en la variable `deficion` queda un mensaje anunciando que la palabra no existía.

3.7.3.1. El valor None

El valor `None`, que se puede traducir como `ninguno`, es un valor que se usa en Python para denotar la “ausencia de un valor”. El tipo de `None` es `NoneType` y es ese nombre el que se menciona cuando se hace una operación con este valor:

```
>>> None + 1
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
```

Nosotros ya hemos utilizado `None` para describir el tipo de retorno de una función que no se espera que retorne nada. Ahora vamos a utilizar `None` como un valor y no como un tipo. Este valor es de mucha utilidad cuando el resultado de una operación puede no existir. Considere por ejemplo el caso de las soluciones reales de una ecuación cuadrática: la ecuación puede tener dos soluciones diferentes, dos soluciones iguales, o ninguna solución [4]. Lo normal sería que, en este último caso, se usara `None` como valor para las soluciones.

```

import math
def solucionar_cuadratica(a: int, b: int, c:int) -> tuple:
    """ Encuentra las soluciones reales de una ecuación cuadrática de la forma
        y = ax^2 + bx + c
    Parámetros:
        a (int): El coeficiente del término de orden 2
        b (int): El coeficiente del término de orden 1
        c (int): El coeficiente del término de orden 0
    Retorna:
        (tuple): Una tupla con las soluciones reales de la ecuación.
        Retorna None si la ecuación no tiene solución real.
    """
    soluciones = None
    determinante = (b**2) - (4*a*c)
    if determinante >= 0:
        sol1 = -b + (math.sqrt(determinante))
        sol2 = -b - (math.sqrt(determinante))
        soluciones = (sol1, sol2)
    return soluciones

def imprimir_soluciones(soluciones: tuple) -> None:
    """ Imprime las soluciones de una ecuación cuadrática o imprime
        un mensaje indicando que no había soluciones.
    Parámetros:
        soluciones (tuple): Una tupla con dos elementos que son las soluciones de la ecuación.
        Si no hay soluciones reales, 'soluciones' debe tener el valor None.
    """
    if soluciones is None:
        print("La ecuación no tenía soluciones reales")
    else:
        print("Las soluciones son", soluciones[0], "y", soluciones[1])

# Calcular e imprimir las soluciones de una ecuación sin soluciones reales
soluciones = solucionar_cuadratica(1,1,1)
imprimir_soluciones(soluciones)

# Calcular e imprimir las soluciones de una ecuación con dos soluciones reales diferentes
soluciones = solucionar_cuadratica(1,0,-1)
imprimir_soluciones(soluciones)

```

En este fragmento tenemos una función llamada `solucionar_cuadratica` que recibe los coeficientes de una ecuación cuadrática y calcula una `tupla` con una pareja de soluciones. Por ahora no se preocupe por la tupla sino por el hecho de que, si la ecuación no tiene soluciones, la función retorna el valor `None`.

La segunda función, `imprimir_soluciones` espera una tupla con dos soluciones en una tupla o el valor `None`. Para saber qué mensaje imprimir, la función usa la condición `soluciones is None`. En general `x is None` es la expresión preferida en Python para saber si alguna variable tiene el valor `None`.

Veamos ahora el resultado de correr el programa anterior:

```

La ecuación no tenía soluciones reales
Las soluciones son 2.0 y -2.0

```

💡 Tip

Use “`is None`” Cuando vaya a revisar si un valor es `None`, use la expresión `is None` en lugar de una comparación (`'== None'`).

3.7.3.2. El método get y el valor `None`

La explicación anterior sobre el valor `None` es relevante porque se usa muy frecuentemente con el método `get`: si una llave no se encuentra en un diccionario, usar `None` como valor por defecto es lo más natural en muchos casos y es mucho mejor que usar cosas como cadenas vacías o sólo con espacios. Observemos el uso de `get` y `None` en un ejemplo:

```
def imprimir_definicion(diccionario: dict, palabra: str) -> None:
    """ Imprime la definición de una palabra o, si la palabra no existe,
    un mensaje indicando el problema.
    Parámetros:
        diccionario (dict): Un diccionario con las palabras y sus definiciones
        palabra (str): La palabra para la que se quiere la definición
    """
    definicion = palabras.get(palabra, None)
    if definicion is not None:
        print("La definición de", palabra, "es:", definicion)
    else:
        print("La palabra '" + palabra + "' no se encuentra en el diccionario")
```

3.7.4. Modificación de diccionarios

Ya vimos cómo implementar en Python los dos usos que usualmente le damos al diccionario de un idioma como el Español. Veamos ahora cómo harían los miembros de la RAE para modificar el diccionario agregando nuevos términos y definiciones y eliminando términos en desuso.

Para explicar cómo se modifica un diccionario, y mostrar de paso que un diccionario se utiliza igual que cualquier otra variable, vamos a crear una nueva función que agrega definiciones al diccionario. Empezaremos con una versión sencilla y la iremos volviendo progresivamente más compleja.

3.7.4.1. Agregar una definición: primera versión

```
def agregar_definicion(diccionario: dict, palabra: str, definicion: str)-> None:
    diccionario[palabra] = definicion
```

Esta primera versión de la función muestra cómo se modifica un diccionario: usando la misma convención que para consultar (paréntesis cuadrados alrededor del nombre de la llave), le *asignamos* una definición a la palabra.

Veamos ahora cómo invocar la nueva función:

```
palabras = {}
agregar_definicion(palabras, 'imagen', 'Figura, representación, semejanza y apariencia de algo')
agregar_definicion(palabras, 'figura', 'Forma exterior de alguien o de algo')
```

Después de ejecutar estas 3 instrucciones, vamos a tener un nuevo diccionario llamado `palabras` que va a tener las definiciones de `imagen` y `figura`.

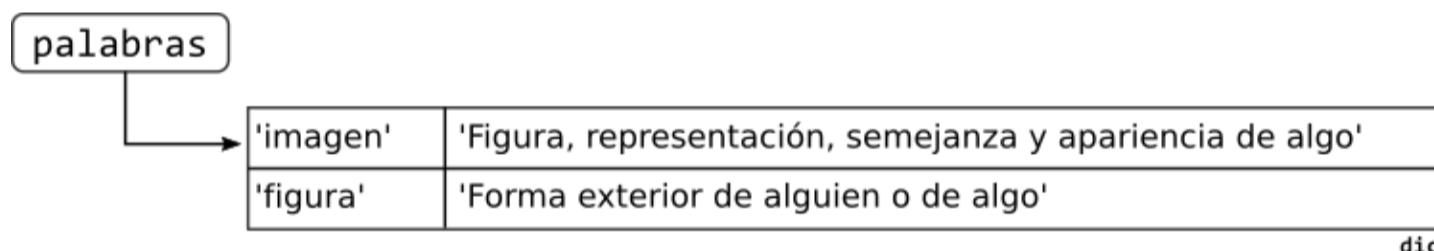


Fig. 3.4 Representación gráfica del diccionario `palabras` (2).

La pregunta que debería surgir en este punto es: ¿Por qué se modifica el diccionario `palabras` si la función no retorna nada? La respuesta a esta pregunta la revisaremos en una sección posterior en la que entraremos en detalle sobre la mutabilidad de los diccionarios.

3.7.4.2. Agregar una definición: segunda versión

Para la segunda versión de la función, vamos a cambiar la firma para que nuestra función retorne un valor de verdad indicando si se pudo agregar la definición o no. Si la palabra ya existía en el diccionario, no debería agregarse y la función debería retornar el valor `False`. De lo contrario, debería agregarse y el resultado debería ser `True`.

```
def agregar_definicion(diccionario: dict, palabra: str, definicion: str)-> bool:
    definicion_agregada = False
    if palabra not in diccionario:
        diccionario[palabra] = definicion
        definicion_agregada = True
    return definicion_agregada
```

Observe 3 cosas interesantes en esta función:

1. La variable `definicion_agregada` se inicializa en `False`.
2. El valor de la variable `definicion_agregada` sólo se cambia por `True` si la palabra se pudo agregar.
3. Usamos el operador `not in` para consultar si la llave no estaba en el diccionario.

```
palabras = {}
res1 = agregar_definicion(palabras, 'imagen', 'Figura, representación, semejanza y apariencia de algo')
res2 = agregar_definicion(palabras, 'figura', 'Forma exterior de alguien o de algo')
res3 = agregar_definicion(palabras, 'imagen', 'Figura, representación, semejanza y apariencia de algo')
print(res1, res2, res3)
```

En este fragmento usamos la nueva función y guardamos el resultado de cada invocación en una variable. Cuando imprimimos las tres variables el resultado que vemos en la consola es `True True False`. Esto nos indica que las dos primeras invocaciones fueron exitosas y que la tercera falló.

3.7.4.3. Agregar una definición: tercera versión

En la tercera y última versión vamos a ofrecer la posibilidad de tener varias definiciones para una palabra. Para lograr esto vamos a concatenar las definiciones a medida que las vayamos agregando, pero sólo si no habíamos almacenado antes esa misma definición. Para verificar este último punto usaremos la operación `not in` aplicada sobre un `str` (la definición que ya teníamos almacenada).

```
def agregar_definicion(diccionario: dict, palabra: str, definicion: str) -> bool:
    definicion_agregada = False
    # La palabra es nueva en el diccionario
    if palabra not in diccionario:
        diccionario[palabra] = definicion
        definicion_agregada = True
    # La palabra no es nueva pero la definición sí es nueva
    elif definicion not in diccionario[palabra]:
        diccionario[palabra] += '\n' + definicion
        definicion_agregada = True
    return definicion_agregada
```

En esta nueva función tenemos una primera posibilidad y es que la palabra no estuviera en el diccionario. En este caso, la palabra se agrega igual que en el caso anterior.

La segunda posibilidad es que la palabra ya estuviera en el diccionario. En este caso, lo que hacemos es revisar si la definición que queremos agregar es parte de la definición que tenemos en el diccionario. Esto lo hacemos aplicando el operador `not in` sobre la definición que sacamos del diccionario: este operador retornará `True` sólo si la nueva definición no está contenida en la definición que estaba guardada en el diccionario. Note que:

- En la condición del bloque `elif` no es necesario incluir la expresión `palabra in diccionario` porque solamente vamos a evaluar el bloque cuando la condición del `if` sea falsa.
- Como sabemos que la palabra sí está en el diccionario, podemos consultarla en la condición del `elif` sin temor a que se genere un error.

Finalmente, en el cuerpo del bloque `elif` se modifica la definición almacenada en el diccionario. Fíjese que estamos usando el operador `+=` para modificar el valor del diccionario de la misma forma en que lo usaríamos para modificar el valor de una variable.

```
palabras = {}
res1 = agregar_definicion(palabras, 'imagen', 'Figura, representación, semejanza y apariencia de algo')
res2 = agregar_definicion(palabras, 'imagen', 'Estatua, efigie o pintura de una divinidad o de un personaje sagrado.')
res3 = agregar_definicion(palabras, 'imagen', 'Figura, representación, semejanza y apariencia de algo')
print(res1, res2, res3)
```

Si ejecutamos este último fragmento usando la nueva definición de la función encontraremos que el resultado que se imprime en la consola es `True True False`. Esto significa que las dos primeras definiciones se pudieron agregar, mientras que la tercera fue rechazada porque estaba repetida.

3.7.4.4. Eliminar una definición

La última operación para estudiar es la que nos permite eliminar definiciones de un diccionario. Esto se puede lograr de dos formas: con el operador `del` o con el método `pop`. Tenga en cuenta que en ambos casos es necesario verificar que la llave exista en el diccionario antes de intentar eliminarla. De lo contrario se producirá un error.

3.7.4.4.1. Uso del operador `del` para eliminar un valor

En la siguiente función se usa el operador `del` para eliminar una palabra del diccionario. Tenga en cuenta que esto eliminará tanto la palabra como su definición. Para evitar que se produzca un error, el llamado a `del` ocurre dentro del cuerpo de un condicional que se asegura que la palabra sí exista en el diccionario.

```
def eliminar_palabra(diccionario: dict, palabra: str) -> bool:
    palabra_eliminada = False
    if palabra in diccionario:
        del diccionario[palabra]
        palabra_eliminada = True
    return palabra_eliminada
```

En el siguiente fragmento se pone en uso la función para eliminar una palabra que no se encuentre en nuestro diccionario y una que sí lo esté [5].

```
palabras = {}
agregar_definicion(palabras, 'imagen', 'Figura, representación, semejanza y apariencia de algo')
agregar_definicion(palabras, 'toballa', 'Toalla')
agregar_definicion(palabras, 'toballa', 'Pieza de felpa')
res1 = eliminar_palabra(palabras, 'caracter')
res2 = eliminar_palabra(palabras, 'toballa')
print(res1, res2)
```

3.7.4.4.2. Uso del método `pop` para eliminar un valor

En la segunda versión de la función se remplazó el operador `del` por un llamado al método `pop`. Al igual que antes, es necesario verificar que la llave efectivamente exista dentro del diccionario antes de hacer el llamado para que no se produzca ningún error.

```
def eliminar_palabra(diccionario: dict, palabra: str) -> bool:
    palabra_eliminada = False
    if palabra in diccionario:
        diccionario.pop(palabra)
        palabra_eliminada = True
    return palabra_eliminada
```

El método `pop` tiene además un interesante resultado y es que retorna el valor eliminado. La siguiente sería una nueva versión de la función aprovechando esta característica.

```
def eliminar_palabra(diccionario: dict, palabra: str) -> bool:
    palabra_eliminada = False
    if palabra in diccionario:
        definicion_eliminada = diccionario.pop(palabra)
        print("Se eliminó la llave", palabra, "que tenía la definición", definicion_eliminada)
        palabra_eliminada = True
    return palabra_eliminada
```

Nota: recuerde que no es recomendable mezclar las instrucciones de interacción (inputs y prints) con las instrucciones de su programa que manejan la información, hacen cálculos, etc.

3.7.4.5. Eliminar todas las definiciones

Finalmente, Python ofrece una manera para eliminar con facilidad todos los elementos de un diccionario: el método `clear`. Por ejemplo, si queremos eliminar todos los elementos del diccionario `palabras` sólo debemos ejecutar la siguiente instrucción:

```
palabras.clear()
```

3.7.5. Usos de los diccionarios

Ya estudiamos todos los mecanismos para construir diccionarios, consultar y modificar valores, agregar nuevos valores y eliminar valores. Ahora vamos a estudiar algunos usos interesantes de los diccionarios.

3.7.5.1. Histogramas basados en diccionarios

Una problemática relativamente común es la de contar cuántas veces aparecen ciertos valores dentro de algo más grande. Por ejemplo, cuántas veces aparece cada uno de los dígitos entre 0 y 9 dentro de un número entero, cuántas veces aparece cada letra dentro de una palabra, o cuántas veces aparece cada palabra dentro de un texto. Un problema relacionado es el de contar cuántos elementos de un conjunto caen dentro de unas ciertas categorías (ej. cuántas personas de un grupo nacieron en cada uno de los meses del año, cuántos estudiantes aprobaron un curso, cuántos reprobaron, etc.).

Matemáticamente, un histograma le asigna un número a cada uno de los valores posibles de un grupo: el número indica cuántas veces apareció el valor que estábamos contando (dígitos, letras, palabras, meses, etc.).

Gráficamente, un histograma se representa con un gráfico de barras en el cual el tamaño de cada barra es proporcional a la cantidad de veces en que apareció cada uno de los valores.

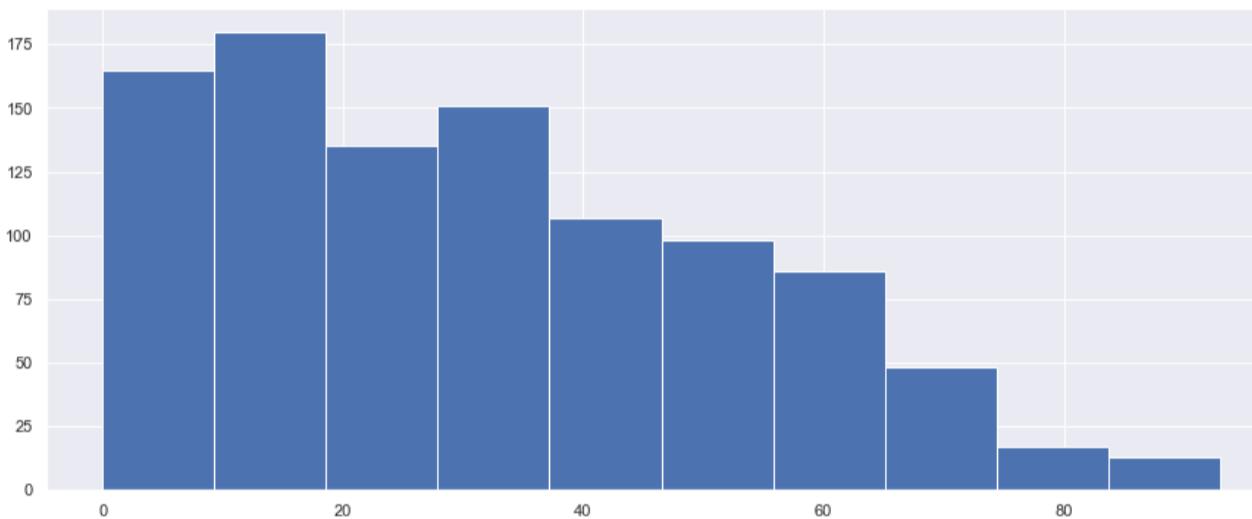


Fig. 3.5 Ejemplo gráfico de un histograma

Los diccionarios se prestan perfectamente para construir histogramas: las llaves serán los valores que queremos contar y los valores del diccionario indicarán la cantidad de veces que apareció cada uno. La siguiente función ilustra esto:

```
def contar_vocales(texto: str) -> dict:
    """ Cuenta la cantidad de veces que aparece cada vocal dentro de un texto
    Parámetros:
        texto (str): El texto en el que van a contarse las vocales
    Retorno:
        (dict): Un diccionario donde las llaves son las vocales minúsculas y los valores
        son la cantidad de veces que aparece la vocal dentro del texto.
    """
    histograma = {}
    histograma['a'] = texto.lower().count('a')
    histograma['e'] = texto.lower().count('e')
    histograma['i'] = texto.lower().count('i')
    histograma['o'] = texto.lower().count('o')
    histograma['u'] = texto.lower().count('u')
    return histograma
```

Esta función primero crea un histograma vacío y luego agrega una nueva llave para cada una de las vocales. El valor asociado a cada llave es el resultado de llamar al método `count()` de `str`, usando como parámetros la vocal correspondiente.

En el siguiente fragmento de código puede apreciarse el resultado de invocar la función usando como parámetro un texto en español bien conocido por ser un pangrama: una frase que usa todas las letras del alfabeto.

```
>>> pangrama = 'Jovencillo emponzoñado de whisky, ¡qué figurota exhibe!'
>>> vocales = contar_vocales(pangrama)
>>> print(vocales)
{'a': 2, 'e': 5, 'i': 4, 'o': 6, 'u': 2}
```

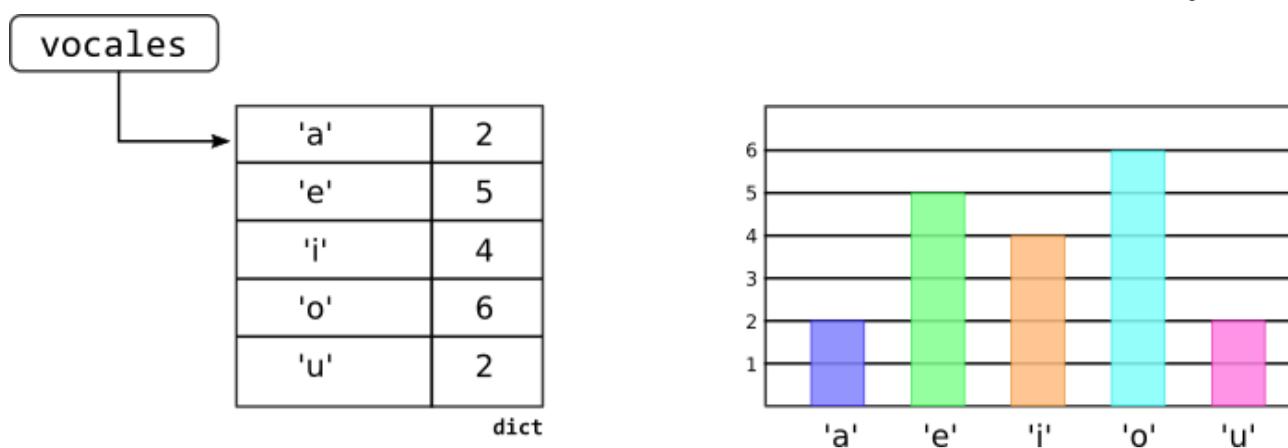


Fig. 3.6 Diccionario con el histograma y representación gráfica

3.7.5.2. Diccionarios como conjuntos

Un segundo uso posible de los diccionarios es representar conjuntos [6]. En un conjunto, cada valor puede aparecer máximo una vez y la pregunta más interesante es si un valor pertenece o no pertenece al conjunto. Cuando un conjunto se representa usando un diccionario, sólo nos interesan las llaves y no los valores.

El siguiente ejemplo ilustrará este punto usando números como llaves en un diccionario y el valor booleano `True` como único valor del diccionario.

```

import random
def lanzar_dado(resultados: dict) -> None:
    """ Lanza un dado calculando aleatoriamente un número entre 1 y 6.
    Registra en el diccionario 'resultados' el valor que se obtuvo, asignándole
    el valor True a la llave que corresponde al valor.
    Parámetros:
        resultados (dict): Un diccionario que representa el conjunto de valores diferentes
            que se han obtenido en los lanzamientos pasados.
    """
    valor = random.randint(1,6)
    resultados[valor] = True

# Lanzar el dado 6 veces y registrar los resultados obtenidos
def lanzar_6_dados() -> dict:
    """ Lanza el dado 6 veces y retorna un diccionario con los valores que se obtuvieron
    Resultado:
        (dict): Un diccionario donde sólo aparecen como llaves los valores que se
            obtuvieron en el lanzamiento del dado.
    """
    resultados = {}
    lanzar_dado(resultados)
    lanzar_dado(resultados)
    lanzar_dado(resultados)
    lanzar_dado(resultados)
    lanzar_dado(resultados)
    lanzar_dado(resultados)
    return resultados

def contar_resultados_diferentes(resultados: dict) -> int:
    """ Cuenta cuántos resultados diferentes hubo
    Parámetros:
        resultados (dict): El conjunto de los resultados obtenidos representado
            utilizando un diccionario. Si un valor aparece como
            llave en el diccionario, significa que el valor se
            obtuvo en el lanzamiento de los dados.
    Retorno:
        (int): La cantidad de resultados diferentes que hubo
    """
    diferentes = 0
    if 1 in resultados:
        diferentes += 1
    if 2 in resultados:
        diferentes += 1
    if 3 in resultados:
        diferentes += 1
    if 4 in resultados:
        diferentes += 1
    if 5 in resultados:
        diferentes += 1
    if 6 in resultados:
        diferentes += 1
    return diferentes

# Lanzar el dado 6 veces y registrar los resultados obtenidos
resultados = lanzar_6_dados()
# Contar cuántos valores diferentes se obtuvieron
diferentes = contar_resultados_diferentes(resultados)
print("En 6 lanzamientos del dado se obtuvieron", diferentes, "valores diferentes")

```

El centro del programa es la función `lanzar_dado` la cual calcula aleatoriamente un valor entero entre 1 y 6. Este valor se almacena en el conjunto `resultados`, el cual está representado por un diccionario: cada vez que se lanza el dado, se agrega el valor obtenido al conjunto creando una nueva llave con el valor asociado True. Note que si el mismo valor aparece varias veces, en el diccionario sólo aparecerá una vez, puesto que no puede haber dos llaves iguales.

La función `lanzar_6_dados` crea un conjunto vacío (un diccionario) y llama 6 veces a la función `lanzar_dado`, logrando que en el conjunto `resultado` queden todos los valores que salieron al menos una vez. Como el diccionario inicialmente estaba vacío, los valores que no hayan salido en el dado no aparecerán en el diccionario.

La función `contar_resultados_diferentes` revisa cuáles de los números entre 1 y 6 aparecen en el diccionario usando el operador `in` y retorna la cantidad. Por fuera de las funciones se llama a las últimas dos funciones y finalmente se imprime un mensaje informando cuántos valores diferentes se encontraron. Note que la tercera función podría haberse implementado fácilmente aplicando la función `len` sobre el conjunto para saber cuántos valores diferentes quedaron en este.

3.7.5.3. Diccionarios como estructuras

Un tercer uso posible de los diccionarios es modelar elementos de la realidad que tienen estructuras complejas y que además deben tener la misma estructura. Por ejemplo, al principio del capítulo hablamos de celulares que se describen con la velocidad del procesador, cantidad de memoria, calidad de la cámara, tecnología de la pantalla, tamaño de la pantalla, capacidad de la pila, sistema operativo y versión del sistema operativo. Si queremos manejar la información de muchos celulares, tiene sentido organizar la información de cada uno dentro de un diccionario. Con esto nuestros programas quedarán mejor organizados, necesitaremos menos variables, y nos podremos asegurar de que no nos haga falta ningún atributo para un celular.

Para empezar a modelar nuestros celulares, listaremos sus características y le daremos un nombre sencillo a cada una:

- procesador: velocidad del procesador en GHz.
- memoria: cantidad de memoria en GB.
- camara: calidad de la cámara en mega-pixeles.
- pantalla: tecnología de la pantalla.
- ancho: ancho de la pantalla en pixeles.
- alto: alto de la pantalla en pixeles.
- pila: capacidad de la pila en miliamperios (mAh).
- sistema: nombre del sistema operativo.
- version: versión del sistema operativo.

A continuación, crearemos una función capaz de crear consistentemente diccionarios con estas llaves:

```
def crear_celular(procesador: float, memoria: float, camara: float,
                  pantalla: str, ancho: int, alto: int, pila: float,
                  sistema: str, version: str) -> dict:
    nuevo_celular = {}
    nuevo_celular['procesador'] = procesador
    nuevo_celular['memoria'] = memoria
    nuevo_celular['camara'] = camara
    nuevo_celular['pantalla'] = pantalla
    nuevo_celular['ancho'] = ancho
    nuevo_celular['alto'] = alto
    nuevo_celular['pila'] = pila
    nuevo_celular['sistema'] = sistema
    nuevo_celular['version'] = version
    return nuevo_celular
```

Esta función en realidad no tiene nada que no hayamos estudiado antes en esta sección: crea un nuevo diccionario y guarda los valores recibidos en las posiciones correspondientes del diccionario. En el siguiente fragmento usamos esta función para crear con facilidad cuatro celulares que se almacenan en diccionarios usando la misma estructura.

```
cel1 = crear_celular(2.4, 64, 48, 'OLED', 1080, 2400, 5000, 'Android', '10')
cel2 = crear_celular(2.2, 64, 32, 'OLED', 768, 1080, 3500, 'Android', '8.1')
cel3 = crear_celular(2.0, 32, 18, 'Retina', 375, 812, 4200, 'iOS', '9.0')
cel4 = crear_celular(1.8, 16, 6, 'Retina', 375, 667, 4150, 'iOS', '8.1.4')
```

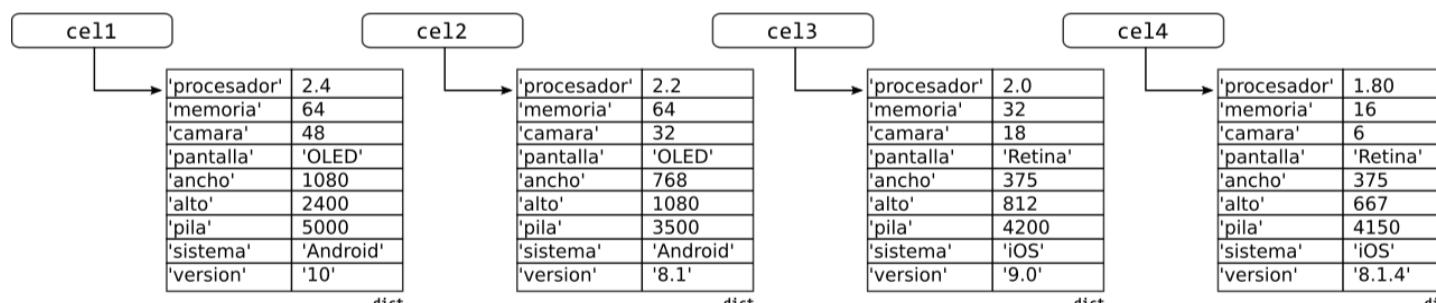


Fig. 3.7 Representación gráfica de los 4 diccionarios para representar celulares

Ahora bien, lo realmente interesante es que ahora podemos construir funciones que conozcan la estructura de estos diccionarios y hagan operaciones sobre los celulares aprovechandose de esta información. Por ejemplo, la siguiente función nos permite comparar los dos celulares para saber cuál tiene la mejor cámara:

```
def mejor_camara(celular1: dict, celular2: dict)-> int:
    """ Busca cuál de los dos celulares tiene la mejor cámara (más mega-pixeles).
    Para que se pueda usar la función, los celulares deben representarse con
    diccionarios que tengan una llave llamada 'camara' que indique la cantidad
    de mega-pixeles de la cámara del celular.
    Parámetros:
        celular1 (dict): Es un diccionario que representa al primer celular.
        celular2 (dict): Es un diccionario que representa al segundo celular.
    Retorno:
        (int): Retorna 1 si el primer celular tiene la mejor cámara.
        Retorna 2 si el segundo celular tiene la mejor cámara.
        Retorna 0 si las cámaras de los dos celulares son iguales.
    """
    mejor = 0
    camara1 = celular1['camara']
    camara2 = celular2['camara']
    if camara1 > camara2:
        mejor = 1
    elif camara1 < camara2:
        mejor = 2
    return mejor
```

Teniendo en cuenta que ya creamos cuatro celulares usando nuestra función constructora (`cel1`, `cel2`, `cel3` y `cel4`) y que esa función nos garantiza la estructura de los diccionarios, podemos usar la nueva función como en el siguiente fragmento:

```
>>> mejor_camara(cel1, cel2)
1
>>> mejor_camara(cel1, cel1)
0
>>> mejor_camara(cel3, cel2)
2
```

A continuación, construiremos una función que es capaz de identificar si alguno de los cuatro celulares tiene una versión determinada de un sistema operativo:

```
def hay_celular_version_so(cel1: dict, cel2: dict, cel3: dict, cel4: dict, so: str, version: str)
-> bool:
    """ Esta función indica si hay algún celular con la versión del sistema operativo
    indicada en los parámetros 'so' y 'version'.
    La función espera que los diccionarios de los celulares tengan una llave llamada
    'sistema' con el nombre del sistema operativo y una llave llamada 'version' con
    la versión del sistema.
    Parámetros:
        cel1 (dict): El diccionario que representa el primer celular
        cel2 (dict): El diccionario que representa el segundo celular
        cel3 (dict): El diccionario que representa el tercer celular
        cel4 (dict): El diccionario que representa el cuarto celular
    Retorno:
        (bool) : Retorna True si algún celular tiene exactamente el mismo sistema operativo
        en la misma versión que se pide en los parámetros 'so' y 'version'.
        Retorna False de lo contrario.
    """
    hay_celular_buscado = False
    if cel1['sistema'] == so and cel1['version'] == version:
        hay_celular_buscado = True
    elif cel2['sistema'] == so and cel2['version'] == version:
        hay_celular_buscado = True
    elif cel3['sistema'] == so and cel3['version'] == version:
        hay_celular_buscado = True
    elif cel4['sistema'] == so and cel4['version'] == version:
        hay_celular_buscado = True
    return hay_celular_buscado
```

Observe que en esta función hemos usado un solo `if` seguido de varios `elif`. Esto ocurre porque una vez se encuentra un celular con el sistema operativo indicado, no es necesario seguir revisando los siguientes. Esto cambiaría si, en lugar de consultar si hay algún celular con las características descritas, quisieramos contar cuántos celulares tienen las características descritas.

```
def contar_celulares_version_so(cel1: dict, cel2: dict, cel3: dict, cel4: dict, so: str, version: str) -> int:
    """ Esta función cuenta cuántos celulares tienen la versión del sistema operativo
        indicada en los parámetros 'so' y 'version'.
        La función espera que los diccionarios de los celulares tengan una llave llamada
        'sistema' con el nombre del sistema operativo y una llave llamada 'version' con
        la versión del sistema.

    Parámetros:
        cel1 (dict): El diccionario que representa el primer celular
        cel2 (dict): El diccionario que representa el segundo celular
        cel3 (dict): El diccionario que representa el tercer celular
        cel4 (dict): El diccionario que representa el cuarto celular
    Retorno:
        (bool) : Retorna la cantidad de celulares que tienen exactamente el mismo
        sistema operativo en la misma versión que se pide en los
        parámetros 'so' y 'version'.
    """
    cantidad_celulares = 0
    if cel1['sistema'] == so and cel1['version'] == version:
        cantidad_celulares += 1
    if cel2['sistema'] == so and cel2['version'] == version:
        cantidad_celulares += 1
    if cel3['sistema'] == so and cel3['version'] == version:
        cantidad_celulares += 1
    if cel4['sistema'] == so and cel4['version'] == version:
        cantidad_celulares += 1
    return cantidad_celulares
```

3.7.5.4. Diccionarios de diccionarios

Por último, vamos a describir un uso de los diccionarios que empieza a mostrar el verdadero poder de estas estructuras de datos. Hasta el momento, los valores que hemos introducido dentro de los diccionarios han sido tipos simples: `int`, `str`, `float` y `bool`. Ahora vamos a introducir también *diccionarios* dentro de los *diccionarios*.

Para ilustrar este punto vamos a continuar con el ejemplo de los celulares y construir un diccionario que va a contener todos los celulares que queremos comparar. En este diccionario las *llaves* serán los *nombres de los celulares* y los *valores* serán los *diccionarios con el resto de características*.

Suponiendo que ya tenemos nuestros cuatro celulares (`cel1`, `cel2`, `cel3` y `cel4`), podemos armar nuestro diccionario de celulares usando las siguientes instrucciones:

```
celulares = {}
celulares['AmazingCel'] = cel1
celulares['BoringCel'] = cel2
celulares['CheapCel'] = cel3
celulares['DumbCel'] = cel4
```

La siguiente imagen muestra gráficamente el resultado de esta ejecución. En la parte superior podemos ver las cuatro variables anteriores apuntando a cada uno de los diccionarios que representan celulares. En la parte inferior podemos ver que los valores en el nuevo diccionario `celulares` apuntan a los mismos diccionarios que identificamos con las variables.

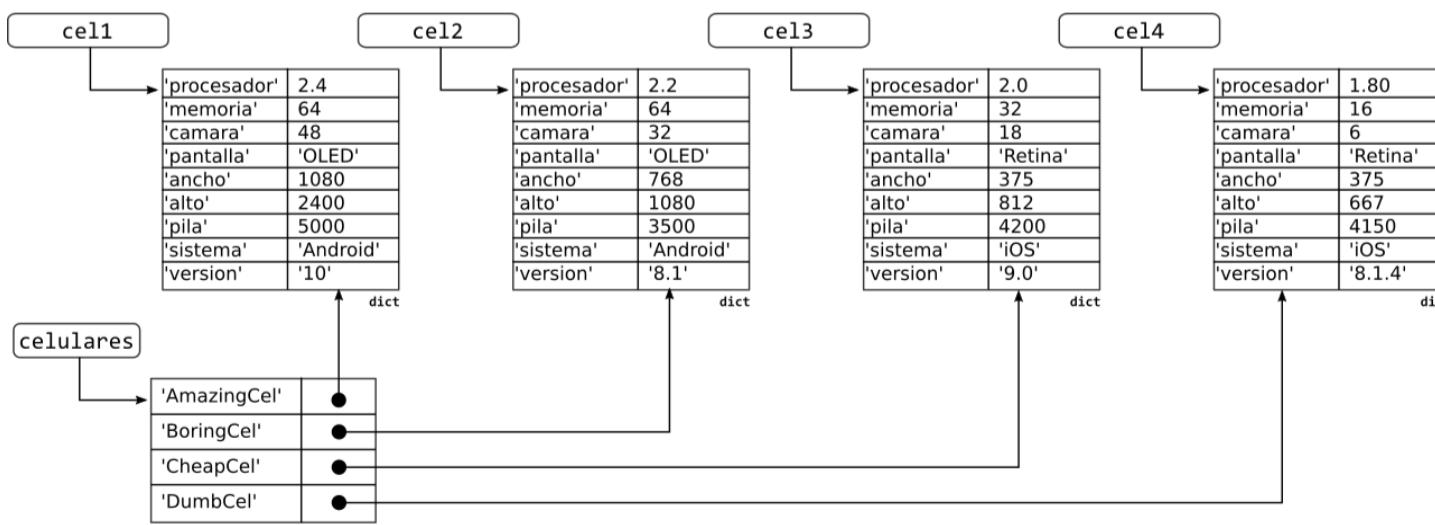


Fig. 3.8 Representación gráfica del diccionario de celulares

Observe ahora lo que pasa si intentamos extraer un valor del diccionario `celulares` usando el nombre de un celular (los cambios de línea los agregamos para facilitar la lectura):

```
>>> print(celulares['AmazingCel'])
{'procesador': 2.4, 'memoria': 64, 'camara': 48,
 'pantalla': 'OLED', 'ancho': 1080, 'alto': 2400,
 'pila': 5000, 'sistema': 'Android', 'version': '10'}
```

Más interesante aún es lo que pasa si encadenamos los llamados para extraer una propiedad del celular:

```
>>> print(celulares['AmazingCel']['procesador'])
2.4
```

Vamos a construir ahora una función similar a la que construimos antes para comparar dos celulares. La principal diferencia es que ahora usaremos nuestro nuevo diccionario de celulares y los nombres de los celulares que queremos comparar.

```
def mejor_camara_con_nombres(celulares: dict, nombre1: str, nombre2: dict) -> str:
    """ Busca cuál de los dos celulares tiene la mejor cámara
    Parámetros:
        celulares (dict): Un diccionario donde las llaves son los nombres de los celulares
                           y los valores son diccionarios que representan celulares
        nombre1 (str): El nombre del primer celular que se quiere comparar
        nombre2 (str): El nombre del segundo celular que se quiere comparar
    Retorno:
        (str): Retorna el nombre del celular que tiene la mejor cámara o "Empate" si
               las cámaras de los dos celulares son iguales.
        Si sólo uno de los nombres corresponde al de un celular, retorna ese nombre.
        Si ningún nombre corresponde al de un celular, retorna "Nombres inválidos".
    """
    # Extraer los celulares del diccionario usando su nombre
    celular1 = celulares.get(nombre1, None)
    celular2 = celulares.get(nombre2, None)
    # Ninguno de los dos nombres era correcto
    if celular1 is None and celular2 is None:
        nombre_mejor = "Nombres inválidos"
    # Sólo el segundo nombre era correcto
    elif celular1 is None and celular2 is not None:
        nombre_mejor = nombre2
    # Sólo el primer nombre era correcto
    elif celular1 is not None and celular2 is None:
        nombre_mejor = nombre1
    # Los dos nombres eran correctos así que hay que comparar los celulares
    else:
        nombre_mejor = "Empate"
        numero_mejor = mejor_camara(celular1, celular2)
        if numero_mejor == 1:
            nombre_mejor = nombre1
        elif numero_mejor == 2:
            nombre_mejor = nombre2
    return nombre_mejor
```

La nueva función recibe el diccionario con todos los celulares y el nombre de los dos celulares que se quieren comparar. Lo primero que hace es intentar extraer los diccionarios que representan a cada uno de los celulares, usando sus nombres. Para esto se utiliza el llamado `celulares.get(nombre1, None)`: si `nombre1` no coincide con el nombre de ningún celular, el resultado returned es el valor nulo `None`. Si ninguno de los dos nombres era correcto, `celular1` y `celular2` tendrán el valor `None` y la función retornará la cadena “Nombres inválidos”. Si sólo uno de los dos nombres es válido, entonces se retornará el celular al que corresponda el nombre válido. Finalmente, si los dos nombres son válidos, se usará la función `mejor_camara` para calcular cuál de los dos celulares es mejor.

Veamos ahora cómo se usaría esta función:

```
>>> print(mejor_camara_con_nombres(celulares, 'DumbCel', 'BoringCel'))
BoringCel
>>> print(mejor_camara_con_nombres(celulares, 'DumbCel', 'FalseCel'))
DumbCel
>>> print(mejor_camara_con_nombres(celulares, 'FalsestCel', 'FalseCel'))
Nombres inválidos
>>> print(mejor_camara_con_nombres(celulares, 'CheapCel', 'CheapCel'))
Empate
```

Por ahora no vamos a ahondar más en el tema de los diccionarios dentro de otros diccionarios. Para poder explotar al máximo el poder que ofrecen tenemos que esperar a introducir los conceptos principales de recorrido de estructuras de datos (Nivel 3). Cuando hayamos pasado ese punto, retomaremos la discusión.

Warning

¡Cuidado! Los diccionarios sólo pueden usarse como valores dentro de otros diccionarios. NO pueden usarse como llaves.

3.7.6. Mutabilidad en diccionarios

Para cerrar este capítulo sobre diccionarios vamos a volver a la discusión que habíamos iniciado sobre la *mutabilidad* de los diccionarios. A diferencia de otros tipos de datos, los diccionarios pueden cambiar. Esto significa que, cuando se agrega, se modifica o se elimina una llave de un diccionario, el diccionario se modifica pero sigue siendo *el mismo diccionario*.

Esta característica es muy importante y hay que tenerla muy en cuenta cuando se utilizan diccionarios en los parámetros de una función. En síntesis, el hecho de que los diccionarios sean mutables hace que todos los cambios que se hacen a un diccionario que llegue a una función como un parámetro, se vean reflejados también fuera de la función.

Veamos esto con un ejemplo sencillo:

```
def agregar_definicion(diccionario: dict, palabra: str, definicion: str)-> None:
    diccionario[palabra] = definicion
```

Esta función ya la habíamos estudiado y lo único que hace es agregar (o reemplazar) una llave dentro del diccionario. Ahora veamos lo que sucede cuando se invoca esta función y se imprime su contenido:

```
>>> palabras = {}
>>> agregar_definicion(palabras, 'palabra1', 'definicion1')
>>> print(palabras)
{'palabra1': 'definicion1'}
>>> agregar_definicion(palabras, 'palabra2', 'definicion2')
>>> print(palabras)
{'palabra1': 'definicion1', 'palabra2': 'definicion2'}
```

El punto central acá es que en todo el ejemplo sólo hay un diccionario. Cuando se invoca la función `agregar_definicion` y se pasa el diccionario `palabras` como parámetro, es exactamente el mismo diccionario el que se recibe y se modifica. Este comportamiento es propio de los diccionarios (y de otros tipos de datos que estudiaremos más adelante), se conoce como *paso de parámetros por referencia*.

Esto no ocurre con tipos como `int` o `str`, en los cuales el paso de parámetros se hace por valor. Es decir, el valor que se recibe dentro de la función es idéntico al que se usó como parámetro, pero no es el mismo (es una copia, no una referencia). Esto se muestra en el siguiente ejemplo:

```
def modificar(entero: int, cadena: str)->bool:
    entero += 100
    cadena += '!!!!'
    print(entero, cadena)
    return True
numero = 1
texto = "Hola Mundo"
resultado = modificar(numero, texto)
print(numero, texto)
```

En este caso, se ha definido una función que modifica los parámetros que recibe e imprime los valores modificados. Por fuera de la función, se crean dos variables que luego se usan para invocar a la función. Finalmente, se imprimen las variables que se usaron para invocar la función. El resultado de la ejecución son las siguientes dos líneas:

```
101 Hola Mundo!!!!
1 Hola Mundo
```

La primera línea se imprime desde adentro de la función y demuestra que los parámetros `entero` y `cadena` se modificaron dentro de la función. La segunda línea muestra que las variables con las que se invocó la función, `numero` y `texto`, no sufrieron ningún cambio.

3.7.6.1. Copias de diccionarios

Finalmente, vamos a desarrollar una función en la que no queremos que se modifique el diccionario original, sino que queremos que se construya uno nuevo. La nueva función `agregar_definicion_con_copia` logra este resultado y retorna un diccionario con todos los elementos que tenía el diccionario original y también con el elemento adicional que se está agregando.

```
def agregar_definicion_con_copia(diccionario: dict, palabra: str, definicion: str)-> dict:
    copia = diccionario.copy()
    copia[palabra] = definicion
    return copia
```

El punto importante dentro de este ejemplo es el uso del método `copy()`, el cual realiza una copia completa del diccionario cuando es invocado. De esta manera, el diccionario que se modifica no es el que llegó como parámetro, sino una copia de este. Esto se ve claramente al ejecutar la nueva función:

```
>>> palabras = {'palabra1': 'definicion1', 'palabra2': 'definicion2'}
>>> copia_palabras = agregar_definicion_con_copia(palabras, 'p99', 'def99')
>>> palabras
{'palabra1': 'definicion1', 'palabra2': 'definicion2'}
>>> copia_palabras
{'palabra1': 'definicion1', 'palabra2': 'definicion2', 'p99': 'def99'}
```

Warning

¡Cuidado! El método `copy()` hace sólo una copia *superficial* del diccionario en lugar de una copia profunda. Esto significa que dentro del nuevo diccionario habrá copias de todos los elementos inmutables, pero habrá referencias a los mismos elementos mutables. En particular, si se hace una copia de un diccionario con diccionarios por dentro, el nuevo diccionario tendrá los mismos diccionarios en lugar de copias también de esos diccionarios.

3.7.7. Ejercicios

1. Construya un diccionario con las alturas sobre el nivel del mar de las capitales de los países suramericanos. Use los nombres de las ciudades, usando mayúsculas al principio de las palabras, como llaves y las alturas como valores. Es decir, las llaves deben ser 'Bogotá', 'Buenos Aires'. etc.
2. Escriba una función que reciba el diccionario con las alturas de las ciudades y los nombres de dos ciudades y que retorne el nombre de la ciudad que esté ubicada a una mayor altura. La función debe ser capaz de funcionar incluso si en el nombre de las ciudades se usan mayúsculas y minúsculas de forma diferente a como están en el diccionario (por ejemplo, 'BoGoTá' y 'BUENOS aires'). Si alguno de los dos nombres no corresponde a una ciudad, la función debe retornar el valor `None`.
3. Escriba una función que reciba un número entero y calcule un diccionario en el que cada llave es un dígito y los valores asociados son la cantidad de veces que aparece el dígito en el número entero. Si un dígito no aparece en el número, no debe aparecer en el diccionario.
4. Escriba una función que reciba un número entero y retorne el dígito que aparece más veces dentro del número.
5. Con base en el ejemplo del diccionario de celulares, construya una función que reciba los diccionarios de 4 celulares y cuente cuántos de estos celulares tienen más de 16 GB de memoria.
6. Con base en el ejemplo del diccionario de celulares, construya una función que reciba los diccionarios de 4 celulares y diga cuál es el celular que tiene la mayor cantidad de pixeles en la pantalla.
7. Con base en el ejemplo del diccionario de celulares, construya una función que reciba los diccionarios de 4 celulares y los modifique para duplicar la capacidad de la pila de todos.
8. Modifique el programa que calcula cuáles valores salieron en los lanzamientos de 6 dados para que ahora calcule un histograma en el que se pueda saber cuántas veces salió cada número.

3.7.8. Más allá de Python

No en todos los lenguajes de programación existe algo equivalente a los diccionarios como un tipo de datos básico. Por ejemplo, en Java es necesario utilizar clases que implementen la interfaz `Map` que es parte del framework de colecciones: aunque es parte de la librería básica de Java, la integración con el lenguaje no es igual de estrecha que en el caso de los diccionarios de Python.

En otros lenguajes, como PHP o Perl hay estructuras de datos prácticamente equivalentes a los diccionarios pero reciben el nombre de *mapas*.

En esta sección estudiamos el uso de los diccionarios como estructuras: esto es una referencia a las estructuras (`struct`) disponibles en C o C++, las cuales sirven para organizar información exactamente igual que como vimos con los diccionarios. Las estructuras pueden considerarse precursoras de los objetos, aunque sin comportamiento asociado.

En esta sección también hizo su aparición el valor `None`: prácticamente todos los lenguajes tienen un valor equivalente, aunque en otros casos se conoce como `null`, `nil` o `NUL`. En general el objetivo es siempre el mismo: especificar de alguna forma que hace falta un valor para algo.

[1] Para el tamaño de la pantalla necesitamos el ancho y el alto.

[2]

Nos referimos a *incómodo* para indicar que, aunque no sería difícil, requeriría un nivel de atención mucho mayor del necesario. Es lo mismo que pasa cuando se usan nombres de variables y parámetros que no corresponden con su significado: no es que sean imposibles de utilizar, pero hacen necesario concentrar la atención en los nombres en lugar de concentrarla en los problemas reales que se están intentando resolver.

- [3] Recordemos que los métodos se invocan usando el operador `.` entre el destinatario del método y el nombre del método.
- [4] El tipo de dato `tuple` sirve para representar secuencias de números, como por ejemplo coordenadas. Lo estudiaremos en más detalle en el nivel 4.
- [5] Para sorpresa de muchos, ‘toballa’ es una palabra que oficialmente hace parte del Español, aunque tal vez debería eliminarse.
- [6] En Python existe también el tipo de dato `set`, que se comporta de forma similar a una lista sin repeticiones. Más adelante se hablará un poco más de este tipo. Por ahora los diccionarios son suficientes para modelar conjuntos.

By Mario Sánchez
© Copyright Agosto de 2020.

[Departamento de Ingeniería de Sistemas y Computación](#) - Universidad de los Andes

3.8. Paso de parámetros

[Print to PDF](#)

 On this page

[3.8.1. Paso de parámetros por valor y por referencia](#)

[3.8.1.1. Parámetros en una función](#)

[3.8.1.2. Parámetros por valor y por referencia](#)

[3.8.1.3. Paso por “referencia al objeto”](#)

[3.8.1.4. Ejercicios](#)

[3.8.2. Parámetros en Python](#)

[3.8.2.1. Parámetros nombrados](#)

[3.8.2.2. Parámetros por defecto / opcionales](#)

[3.8.2.3. Ejercicios](#)

[3.8.3. Más allá de Python](#)

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.

Objetivo de la sección

El objetivo de esta sección es estudiar con un poco más de profundidad el mecanismo de paso de parámetros y presentar además algunas características que tiene el lenguaje Python pero no otros.

Hasta el momento hemos utilizado el mecanismo de paso de parámetros cada vez que hemos invocado una función, pero no hemos entrado en mucho detalle sobre lo que está pasando. En esta sección vamos a estudiar este proceso en más profundidad y vamos a diferenciar entre los dos mecanismos disponibles en Python: *paso de parámetros por valor* y *paso de parámetros por referencia*.

Además, estudiaremos un par características adicionales que ofrece Python para manejar los parámetros de una función con mayor flexibilidad. Usar estas características es totalmente opcional y, como usualmente no están disponibles en otros lenguajes, no es recomendable acostumbrarse a usarlas siempre.

3.8.1. Paso de parámetros por valor y por referencia

Para empezar, vamos a recordar la diferencia entre el operador `==` y el operador `is`: el primero lo podemos usar para comparar los valores de unas variables para ver si son iguales, mientras que el segundo nos sirve para saber si son *el mismo*. Esto lo podemos ver en el siguiente ejemplo en el cual estamos construyendo dos diccionarios y luego los comparamos usando los dos operadores:

```

1  >>> d1 = {"k1" : 1}
2  >>> d2 = {"k1" : 1}
3  >>> d1 == d2
4  True
5  >>> d1 is d2
6  False

```

- En el caso del operador `==`, el valor obtenido es `True` porque los dos diccionarios tienen las mismas llaves y los mismos valores asociados. Se dice entonces que `==` compara los valores.
- En el caso del operador `is`, el valor obtenido es `False` porque estamos hablando de dos diccionarios: en la línea 1 se construye el primero y en la siguiente línea se construye el segundo. Es apenas una casualidad que las llaves y los valores sean los mismos: si revisáramos la memoria del computador, nos daríamos cuenta que cada diccionario ocupa un espacio diferente en la memoria. Se dice entonces que `is` compara las referencias (en este contexto, una referencia sería la dirección de memoria donde se encuentre un diccionario).

Important

El operador `==` se usa para comparar valores.

El operador `is` se usa para comparar referencias.

3.8.1.1. Parámetros en una función

Veamos ahora lo que pasa cuando se tienen parámetros de diferentes tipos en una función. Para esto estudiemos el siguiente programa:

```

1 def funcion(p1: str, p2: str, p3: dict, p4:dict) -> None:
2     print("--> Valores recibidos:", p1, p2, p3, p4)
3     print("--> Comparar las cadenas:", p1 == p2)
4     print("--> Comparar los diccionarios:", p3 == p4, p3 is p4)
5
6     p1 = "nueva cadena"
7     p3["nuevo valor"] = 99
8     print("--> Valores modificados:", p1, p2, p3, p4)
9     p3 = {"ultimo": 1}
10    print("--> Valores modificados de nuevo:", p3, p4)
11
12    cadena = "cadena inicial"
13    diccionario = {"inicial": 0}
14
15    print("Antes de llamar a la función: ", cadena, diccionario)
16    funcion(cadena, cadena, diccionario, diccionario)
17    print("Después de llamar a la función: ", cadena, diccionario)

```

El siguiente bloque muestra el resultado completo de ejecutar el programa.

```

1 Antes de llamar a la función: cadena inicial {'inicial': 0}
2 --> Valores recibidos: cadena inicial cadena inicial {'inicial': 0} {'inicial': 0}
3 --> Comparar las cadenas: True
4 --> Comparar los diccionarios: True True
5 --> Valores modificados: nueva cadena cadena inicial {'inicial': 0, 'nuevo valor': 99}
6 {'inicial': 0, 'nuevo valor': 99}
7 --> Valores modificados de nuevo: {'ultimo': 1} {'inicial': 0, 'nuevo valor': 99}
Después de llamar a la función: cadena inicial {'inicial': 0, 'nuevo valor': 99}

```

Estudiemos ahora, paso por paso, lo que hace el programa. En las líneas 1 a 10 se define una nueva función que no se ejecutará por ahora. En las líneas 12 y 13 se definen dos variables: la primera es de tipo `str` y tiene como valor `"cadena inicial"`; la segunda es de tipo `dict` y tiene como valor un diccionario que únicamente tiene una llave (`"inicial"`) con el valor `0` asociado.

En la 15 se imprime un mensaje en la consola que muestra el valor de las dos variables: `Antes de llamar a la función: cadena inicial {'inicial': 0}`. En el bloque de código que se encuentra más abajo se puede ver todo lo que imprime el programa.

Luego, en la línea 16 se invoca a la función: la variable `cadena` se usa para los parámetros `p1` y `p2`; la variable `diccionario` se asigna a los parámetros `p3` y `p4`.

La función que definimos imprime los parámetros que recibimos (línea 1), imprime el resultado de comparar `p1` y `p2` (línea 3), e imprime el resultado de comparar `p3` y `p4` usando tanto `==` como `is`. Como es de esperarse, en todos los casos el resultado que se muestra en la consola es `True`.

A partir de la la siguiente línea empiezan a ocurrir cosas más interesantes. En la línea 6, se le asigna un nuevo valor a `p1` que, recordemos, es de tipo `str`. En la línea 7 se le asigna también un nuevo valor a la llave `"nuevo_valor"` dentro del diccionario `p3`.

En la línea 8 se imprimen los valores modificados y se obtiene la siguiente cadena: `--> Valores modificados: nueva cadena cadena inicial {'inicial': 0, 'nuevo valor': 99} {'inicial': 0, 'nuevo valor': 99}`:

- En `p1` vemos el nuevo valor que le asignamos en la línea 6.
- En el parámetro `p2` vemos el valor inicial que tenía el parámetro. Esto se explica porque cuando hicimos la asignación Python creó una variable nueva, local a la función, llamada `p1` y con valor inicial `99`. Es decir que `p1` y `p2` ahora van a ser dos elementos independientes.
- En `p3` vemos que tenemos ahora un diccionario que además del valor inicial tiene el nuevo valor que agregamos en la línea 7.
- En `p4` vemos que también tenemos el nuevo valor: en la línea 7 le agregamos una nueva llave al diccionario al que apuntaba la referencia `p3`, pero ese diccionario era *el mismo* al que apuntaba `p4`. Por eso cuando modificamos el diccionario `p3` también se modificó el diccionario que conocemos como `p4` (*es el mismo!*)

A continuación, la línea 9 hace una nueva asignación pero sobre `p3`: se construye un nuevo diccionario con la llave `"ultimo"` y se almacena en `p3`. En la siguiente línea volvemos a imprimir los diccionarios: `--> Valores modificados de nuevo: {'ultimo': 1} {'inicial': 0, 'nuevo valor': 99}`

- En `p3` ahora tenemos sólo el nuevo diccionario (el que tiene sólo la llave llamada `"ultimo"`).
- En `p4` tenemos el diccionario que teníamos en el paso anterior (el que tiene las llaves `"inicial"` y `"nuevo valor"`).

Esto concluye la ejecución de la invocación a la función así que el programa sigue adelante ejecutando la línea 17, en la cual imprime nuevamente las variables `cadena` y `diccionario`, con lo cual obtenemos el siguiente resultado:

Después de llamar a la función: `cadena inicial {'inicial': 0, 'nuevo valor': 99}`.

Este último resultado nos muestra lo siguiente:

- La variable `cadena` sigue teniendo el mismo valor asociado. A pesar de que la función modificó a `p1`, el valor de `cadena` no cambió. Esto pasa porque, para efectos prácticos, `p1` y `p2` eran *copias* de `cadena`.
- La variable `diccionario` muestra que la primera modificación al diccionario `p3` quedó registrada, pero no la segunda. Esto pasa porque el diccionario inicial se pasó como parámetro a la función y nunca se creó una copia de él. Cuando modificamos a `p3` en la línea 7 estábamos modificando el diccionario inicial. Luego, en la línea 9 se creó un nuevo diccionario y ese se asignó a la variable `p3`, pero eso no tiene por qué tener ningún efecto en el diccionario al que señalaba `p4` y mucho menos al que señalaba la variable `diccionario`.

3.8.1.2. Parámetros por valor y por referencia

Después de estudiar el ejemplo anterior, podemos explicar mejor el significado de pasar parámetros por valor y por referencia cuando se invoca una función.

Un parámetro se pasa por valor, cuando el valor se copia para que se utilice dentro de la función. Esto es lo que ocurre con algunos tipos básicos como `int`, `float` y `bool`, y también con tipos inmutables como `str` y `tuple` (que estudiaremos más adelante).

Por otro lado, un parámetro se pasa por referencia, cuando lo que se hace es entregarle a la función una referencia al elemento, sin copiarlo [1]. Esto quiere decir que si la función hace cambios al elemento, esos cambios los podrá ver quien haya invocado a la función. Esto va a ocurrir en Python con tipos de datos como los diccionarios (`dict` que ya estudiamos, y las listas `list` que estudiaremos en el siguiente nivel).

Veamos un último ejemplo para ilustrar ese punto:

```

1 def limpiar_diccionario(el_diccionario: dict) -> None:
2     el_diccionario.clear()
3
4 d = {"a":1, "b":2, "c": 3}
5 print(d)
6 limpiar_diccionario(d)
7 print(d)
```

En este programa se crea un diccionario con 3 elementos y luego se pasa por referencia a la función `limpiar_diccionario`, que llama el método `clear` sobre el diccionario que recibe por parámetro. Cuando imprimimos de nuevo el diccionario `d`, este va a estar vacío porque la función lo limpió. Si los diccionarios se pasaran por valor (es decir que se crearan copias en cada invocación), no veríamos ningún cambio sobre el diccionario después de la invocación.

3.8.1.3. Paso por “referencia al objeto”

La realidad del paso de parámetros en Python es más complicada de lo presentada en las secciones anteriores. En Python, los parámetros se dice que pasan como *referencia a los objetos*. Esto quiere decir que, incluso los tipos que parece que se pasan por valor, se pasan como referencia. Esto tiene como consecuencias principales un minúsculo ahorro en el *uso de la memoria* y una mejora posiblemente imperceptible en el *desempeño* de los programas.

No ahondaremos en la explicación porque no vale la pena entrar en una exposición más compleja sobre la mecánica del paso de parámetros por referencia a objetos, ni sobre las estrategias que utiliza Python para representar datos en memoria, ni mucho menos sobre el hecho de que en Python todo es un *objeto*. La conclusión importante es que para efectos prácticos los valores inmutables se comportan como si se pasaran por valor y los mutables como si se pasaran por referencia.

Important

Los tipos básicos como `int`, `float` y `bool`, y los inmutables como `str` y `tuple` se comportan como si se pasaran por valor.

Los tipos complejos y mutables como `dict` y `list` se comportan como si se pasaran por referencia.

3.8.1.4. Ejercicios

Para cada uno de los ejercicios asegúrese de imprimir los valores que use para sus pruebas antes y después de hacer las invocaciones.

1. Vamos a representar a un estudiante usando un diccionario con 4 llaves: "[nombre](#)", para almacenar su nombre; y "[matemáticas](#)", "[lenguaje](#)" y "[ciencias](#)" para almacenar las notas en cada una de las 3 materias. Escriba una función que reciba un estudiante (un diccionario) y retorne su promedio.
2. Ahora escriba una función que reciba un estudiante (un diccionario), el nombre de una de las tres materias, y una nota, y le asigne al estudiante la nueva nota en esa materia.
3. Escriba una función que reciba tres estudiantes (tres diccionarios) y el nombre de una de las tres materias. La función debe aumentar la nota de cada estudiante en la materia en un 10% del promedio de los tres estudiantes en esa materia.

3.8.2. Parámetros en Python

A continuación estudiaremos dos características adicionales de Python que permiten manejar con un poco más de flexibilidad los parámetros de las funciones que se definen y se invoquen. Estas características usualmente no están disponibles en otros lenguajes.

3.8.2.1. Parámetros nombrados

La primera característica para estudiar tiene que ver con la forma de hacer referencia a los parámetros de una función en el momento de la invocación. Hasta ahora, siempre hemos utilizado el mecanismo basado en la posición: el primer valor en la invocación corresponde al primer parámetro, el segundo valor en la invocación corresponde al segundo parámetro, y así sucesivamente. Sin embargo, en Python es posible hacer explícito el nombre de los parámetros en el momento de la invocación de tal forma que se puedan invocar en un orden diferente al que se tiene en la declaración.

Veamos un ejemplo:

```
def imprimir_nombre(nombre: str, apellido: str) -> None:
    print(nombre + " " + apellido)

imprimir_nombre("Juan", "Perez") # Imprime "Juan Perez"
imprimir_nombre(nombre = "Juan", apellido = "Perez") # Imprime "Juan Perez"
imprimir_nombre(apellido = "Perez", nombre = "Juan") # Imprime "Juan Perez"
```

En este ejemplo hacemos tres invocaciones a la función:

1. En la primera no se usan los nombres de los parámetros, así que la invocación se hace por posición.
2. En la segunda se usan los nombres de los parámetros, en el mismo orden en el que están definidos.
3. En la tercera también se usan los nombres pero un orden diferente al de la definición. En este caso el resultado es el esperado porque gracias al nombre la función puede reconocer a qué parámetro corresponde cada valor.

Veamos ahora unos ejemplos en los que no se utilicen los nombres de todos los parámetros:

```
imprimir_nombre(apellido = "Perez", "Juan") # Falla
imprimir_nombre("Perez", nombre="Juan") # Falla
imprimir_nombre("Juan", apellido = "Perez") # Imprime "Juan Perez"
```

1. En el primer caso, se presentará el error [SyntaxError: positional argument follows keyword argument](#). Este error nos indica que los valores *sin nombre* no deberían ir después de valores con nombre.
2. En el segundo caso, se presentará el error [imprimir_nombre\(\) got multiple values for argument 'nombre'](#). Este error nos indica que Python intentó asignarle el primer valor al primer parámetro ([nombre](#)) y luego le intentó asignar el valor nombrado al mismo parámetro. Como resultado, el parámetro [nombre](#) recibió múltiples valores mientras que [apellido](#) no recibió ninguno.
3. El tercer caso es exitoso: el primer valor se asigna al primer parámetro y el segundo valor se asigna al parámetro [apellido](#). El resultado es que todos los parámetros de la función reciben un valor y por ende se puede hacer la invocación.

El uso de parámetros nombrados no es obligatorio, pero puede ayudar a hacer más legible el código, especialmente cuando no hay un orden en los parámetros que sea fácil de predecir. Por ejemplo, si tuviéramos una función para evaluar un polinomio de la forma $a \cdot x^2 + b \cdot x + c$, la función se invocaría como

`evaluar_polinomio(3, 5, 7)` o como `evaluar_polinomio(7, 5, 3)`? Para evitar la confusión se podría hacer uso de los nombres de los parámetros: `evaluar_polinomio(a=3, b=5, c=7)`.

3.8.2.2. Parámetros por defecto / opcionales

La segunda característica que queremos que usted conozca es la posibilidad de definir valores por defecto para los parámetros. Esto hace posible hacer llamados a funciones utilizando sólo algunos de los parámetros y, en conjunto con los parámetros nombrados, puede hacer que el código de un programa sea mucho más sencillo.

Por ejemplo, veamos la definición de la función `count` de `str` que nos permite contar cuántas veces aparece una subcadena en otra:

```
>>> help(str.count)
S.count(sub[, start[, end]]) -> int
```

Lo que esta definición nos dice es que el parámetro `sub` es obligatorio, y puede estar seguido de un segundo parámetro (`start`), el cual podría estar seguido de un tercer parámetro (`end`). Aunque no lo podemos ver, en esta función `start` y `end` tienen valores por defecto: `start` tiene el valor 0 mientras que `end` tiene un valor igual a la longitud de la cadena. De esta forma, si no se especifica un inicio se empezará desde el primer carácter, y si se especifica un inicio pero no un fin, se irá hasta el final de la cadena.

Veamos ahora un pequeño ejemplo para ilustrar la sintaxis:

```
def replicar(cadena: str, cantidad: int = 2) -> str:
    return cadena * cantidad
```

En este caso, estamos definiendo una función con dos parámetros: el primero es obligatorio, pero el segundo puede hacerse explícito o no. Si no se incluye al hacer una invocación, su valor por defecto será 2. Note que para parámetro `cantidad` se ha indicado primero el tipo (`int`) y luego el valor por defecto.

⚠ Warning

Use valores por defecto para los parámetros únicamente cuando sea evidente cuál debería ser el valor por defecto del parámetro. Si el que invoca la función tiene que pensar mucho sobre el valor por defecto del parámetro, posiblemente haya sido un error ponerle un valor por defecto.

💡 Tip

Use parámetros por defecto para simplificar el llamado a funciones que tengan una gran flexibilidad a través de un gran número de parámetros. Por ejemplo, hay bibliotecas con funciones que esperan más de 10 parámetros pero el hecho de que todos tengan valores por defecto hace que cualquier invocación pueda concentrarse en los parámetros que realmente valgan la pena.

3.8.2.3. Ejercicios

1. Construya una función que reciba dos números enteros como parámetros: si el primer número es par, la función debe retornar la suma de los dos números; de lo contrario, debe retornar la multiplicación de los dos números.
2. Invoca su función con varias parejas de números igual que como lo ha estado haciendo hasta el momento.
3. Invoca la función nuevamente, pero esta vez utilice los nombres de los parámetros: pruebe usando los parámetros en el orden en el que están definidos en la función y también en el orden opuesto. Compruebe que el resultado en cada caso es el que usted esperaría.
4. Invoca la función usando sólo un parámetro. Si no funciona, asegúrese de entender el error que reciba.
5. Modifique la función para que tenga valor por defecto sólo para el segundo parámetro.
6. Repita las invocaciones con uno y con dos parámetros, tanto con los nombres como sin ellos. ¿En qué casos se produjeron errores?
7. Modifique ahora la función para que tenga valor por defecto sólo para el primer parámetro. Vuelva a hacer las pruebas y revise en qué casos se producen errores.

3.8.3. Más allá de Python

A lo largo de los años, diferentes lenguajes han experimentado con diversas mecanismos para el paso de parámetros. Aunque puede parecer un asunto sencillo, escoger un método particular puede tener un impacto muy importante en el desempeño de una aplicación. Por ejemplo, en el caso de Java en todos los llamados los parámetros se pasan por valor: esto quiere decir que siempre se hacen copias, aunque cuando se trata de objetos (tipos no simples) se hacen copias de las referencias a los objetos. Esto le permite a Java tener un mecanismo homogéneo de invocación y paso de parámetros, simplificar el manejo de la pila de ejecución y aumentar la seguridad del lenguaje a través de la protección de los apuntadores.

Tradicionalmente, los valores con los que se invoca una función (o método) se asignan a los parámetros con base en la posición. Esto es algo que viene del cálculo: si tenemos definida la función $f(x, y)$, cuando evaluemos la función (por ejemplo $f(2, 3)$), estamos acostumbrados a que el valor 2 se use para x y el valor 3 se use para y . Sin embargo, el uso de parámetros nombrados ha empezado a hacerse más popular y es una característica de varios lenguajes modernos como Scala o Kotlin.

```
def imprimirNombre(nombre: String, apellido: String): Unit = {
    println(nombre + " " + apellido)
}

imprimirNombre("Juan", "Perez") // Imprime "Juan Perez"
imprimirNombre(nombre = "Juan", apellido = "Perez") // Imprime "Juan Perez"
imprimirNombre(apellido = "Perez", nombre = "Juan") // Imprime "Juan Perez"
```

Finalmente, tenemos a continuación un pequeño ejemplo de Smalltalk, un lenguaje diseñado hace cerca de 40 años. En Smalltalk todos los parámetros son nombrados porque hacen parte del nombre mismo de los métodos. En el siguiente bloque mostramos la definición del método `imprimirNombre:apellido:`, que recibe dos parámetros y luego los imprime como en el ejemplo anterior en Scala.

```
imprimirNombre: elNombre apellido: elApellido

Transcript show: elNombre, ' ', elApellido
```

Con respecto a los valores por defecto para los parámetros, encontramos que es posible asignar valores por defecto en lenguajes como C++ o JavaScript, pero no en Java. Al igual que en Python, se debe tener cuidado de asignarle valores por defecto a los últimos parámetros de una función para que esos parámetros sean realmente opcionales.

- [1] En un lenguaje como C o C++, que tienen acceso de bajo nivel a la memoria, el paso por referencia se basa en el paso de apuntadores. Esto hace posible modificar los apuntadores desde adentro de la función (método), lo cual no es posible en Python. Para no complicar más la explicación, hemos ajustado la definición de paso por referencia a lo que tendría sentido para Python.

By Mario Sánchez
© Copyright Agosto de 2020.

[Departamento de Ingeniería de Sistemas y Computación](#) - Universidad de los Andes

3.9. Errores frecuentes

⚠ 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.

En esta sección presentamos algunos de los errores que encontramos frecuentemente en los programas de muchos estudiantes. Revíselos con cuidado y asegúrese de entenderlos para no cometer esos mismos errores

3.9.1. Confundir los nombres de las variables con los nombres de las llaves en un diccionario

Si utilizamos una variable para indicar una llave en un diccionario, lo único que nos interesa es el valor asociado a la variable y no el nombre utilizado para la variable. Frecuentemente se encuentran confusiones como las del siguiente programa:

```
1 diccionario = {"llave": 1, "mensaje": 2}
2 llave = "mensaje"
3 mensaje = diccionario[llave]
4 print(diccionario[mensaje])
```

En la línea 1, se crea un nuevo diccionario con dos llaves de tipo `str`: `"llave"` y `"mensaje"`. En la línea 2 se crea una variable llamada `llave` cuyo valor es la cadena `"mensaje"`.

En la línea 3 se extrae el valor asociado a la llave almacenada en la variable `llave`. La llave es la cadena `"mensaje"`, así que del diccionario se extrae el valor `1` y se almacena en la variable `mensaje`.

En la línea 4 se intenta extraer el valor asociado a la llave almacenada en la variable `mensaje`. Sin embargo, el valor asociado es el entero `1` y, como esa llave no existe en el diccionario, se produce un error.

3.9.2. Intentar extraer una llave que no existe

Intentar extraer el valor asociado a una llave de un diccionario producirá un error cuando la llave no exista. Las dos formas posibles de evitar esta situación son:

1. Preguntar si la llave existe, usando el operador `in`, antes de intentar el valor.
2. Utilizar el método `get` para no producir el error y obtener en cambio un valor por defecto si la llave no existe en el diccionario.

3.9.3. Reemplazar un valor por accidente

Como en Python se usa la misma sintaxis para asignarle un valor a una llave en un diccionario y para *actualizar* el valor, es fácil reemplazar un valor por accidente. La principal forma de evitar este problema es averiguar si el valor ya existía antes de hacer la modificación utilizando el operador `in`.

Consideremos por ejemplo el siguiente programa donde la función `incrementar_contadores` se utiliza para construir un histograma de valores:

```
def incrementar_contadores(histograma: dict, valor: str) -> None:
    histograma[valor] = 1
```

Esta función siempre le asociará el valor `1` a la llave. En realidad se debería haber preguntado si la llave existía antes de hacer la modificación:

≡ On this page

[3.9.1. Confundir los nombres de las variables con los nombres de las llaves en un diccionario](#)

[3.9.2. Intentar extraer una llave que no existe](#)

[3.9.3. Reemplazar un valor por accidente](#)

[3.9.4. Modificar un diccionario dentro de una función](#)

[3.9.5. Reemplazar un diccionario dentro de la función](#)

[3.9.6. Copias superficiales vs. copias profundas](#)

```
def incrementar_contadores(histograma: dict, valor: str) -> None:
    if valor in histograma:
        histograma[valor] += 1
    else:
        histograma[valor] = 1
```

Otra solución posible se podría construir usando el método `get`:

```
def incrementar_contadores(histograma: dict, valor: str) -> None:
    histograma[valor] = histograma.get(valor, 0) + 1
```

3.9.4. Modificar un diccionario dentro de una función

Cuando se use un diccionario como parámetro de una función, el cuerpo de la función tendrá acceso al diccionario a través del parámetro. Si el diccionario se modifica dentro de la función, estos cambios serán visibles para quien haya hecho la invocación de la función.

Observemos un ejemplo:

```
def funcion(histograma: dict) -> None:
    histograma.clear()
    histograma["llave_inicial"] = 0
```

La función mostrada producirá el siguiente efecto sobre cualquier diccionario que se use como parámetro en una invocación a la función:

1. Eliminará todos los elementos del diccionario
2. Le asignará el valor `0` a la llave `"llave_inicial"`.

3.9.5. Reemplazar un diccionario dentro de la función

Cuando se use un diccionario como parámetro de una función, el cuerpo de la función tendrá acceso al diccionario a través del parámetro. Sin embargo, si se modifica el parámetro reemplazándolo por un nuevo diccionario, el diccionario que se pasó como parámetro no se verá afectado.

Observemos un ejemplo:

```
def funcion(histograma: dict) -> None:
    histograma = {"llave_inicial": 0}
```

En esta función se recibe un diccionario como parámetro, posiblemente inicializado con algunos valores. En la única línea del cuerpo de la función, se crea un nuevo diccionario y se asocia al identificador `histograma`. Aunque parece que estamos reemplazando el valor del parámetro, en realidad estamos creando una nueva variable con el mismo nombre que el parámetro y le estamos asociando como valor un nuevo diccionario.

El diccionario original que se haya usado al invocar la función no se verá afectado de ninguna manera por esta función.

3.9.6. Copias superficiales vs. copias profundas

Cuando se usa el método `copy` de un diccionario, se hace una copia *superficial* en lugar de una copia *profunda*: si el diccionario contenía valores que no fueran valores numéricos (`int`, `float`), booleanos (`bool`) o cadenas (`str`), la copia del diccionario incluirá referencias a los valores originales.

Por ejemplo, si teníamos un diccionario dentro de un diccionario y copiamos el diccionario contenedor, obtendremos un nuevo diccionario contenedor con dentro el mismo diccionario que tenía el contenedor original.

3.10. Para no olvidar

- Use el operador `in` para saber si una llave existe en un diccionario. Es mucho mejor revisar si una llave existe antes de intentar extraerla, que intentar sacar el valor con la llave y obtener un error.
- Use el método `get` para obtener un valor por defecto cuando no exista una llave. Si tiene sentido obtener un valor por defecto, use `get` para extraer un valor de un diccionario.

- Los diccionarios pueden usarse como histogramas. Debido a su eficiencia para encontrar un valor asociado a una llave, los diccionarios son muy útiles para construir histogramas. Además, el uso del método `get` hace que la construcción de un histograma sea muy sencilla.
- Cuando se usen en los parámetros de una función, los diccionarios se pasan por referencia y son mutables. Por este motivo, los cambios que se hagan al diccionario desde adentro de la función serán visibles para el que hizo la invocación.
- El método `copy` crea una copia de un diccionario. Sin embargo debe tener cuidado: será una *copy superficial*.

By Mario Sánchez
© Copyright Agosto de 2020.

[Departamento de Ingeniería de Sistemas y Computación](#) - Universidad de los Andes

Print to PDF ►

3.11. Ejercicios adicionales

[Print to PDF ▶](#)[On this page](#)[3.11.1. Solución de problemas](#)

⚠ 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.

3.11.1. Solución de problemas

1. Enunciado: Durante un juego de cartas, un apostador principiante desea hacer algo de trampa para mejorar su juego. Para esto tiene dos cartas escondidas y podrá tomar una de ellas dependiendo de la carta que ya tenga en la mano. Para escoger la carta con la que hará trampa, el apostador comparará la carta que tiene en la mano ([carta_mano](#)) con las dos opciones de cambio ([opcion_1](#) y [opcion_2](#)). Al apostador le serviría una carta que tenga el mismo número de la carta que tiene en la mano o una que tenga el mismo palo.

- Si las dos cartas escondidas le sirven, el apostador siempre escogerá la primera opción.
- Si ninguna de las dos cartas escondidas tiene el mismo número o el mismo palo que la de la mano, entonces el apostador no hará trampa.

Usted debe construir una función que indique la opción que debería escoger el apostador: [1](#) para la primera carta escondida, [2](#) para la segunda carta escondida, o [0](#) si ninguna carta le ayuda al apostador. Cada carta será representada por un diccionario con las llaves "numero" y "palo". Tenga en cuenta que la llave "numero" también puede tener asociadas las letras '[J](#)', '[Q](#)', '[K](#)' y '[A](#)'.

2. Enunciado: Los cajeros automáticos usualmente tienen restricciones sobre las claves (numéricas) que pueden usarse para retirar. Usted debe construir una función que diga si una clave dada (un número entero con 4 dígitos cuyo primer dígito no es el 0) es permitida de acuerdo a las siguientes reglas:

- El mismo dígito no puede aparecer más de dos veces.
- No puede haber dígitos consecutivos.
- No puede haber números que empiecen por [19](#), [200](#) ni [201](#).
- El número debe tener al menos tres dígitos diferentes.

3. Enunciado: Usted ha sido encargado de escribir una función que decida si una contraseña es lo suficientemente segura para un nuevo sistema. Una contraseña segura debe tener al menos 10 caracteres y cumplir con al menos 3 de las siguientes restricciones adicionales:

- Debe tener al menos una letra mayúscula y una letra minúscula.
- Debe tener al menos un dígito.
- Debe tener al menos uno de los siguientes caracteres: [!@#\\$ %&/\(\)=?#](#)
- No puede empezar ni terminar con un espacio.

4. Enunciado: Escriba una función que reciba 3 diccionarios que representen las coordenadas de 3 puntos en el espacio. Cada diccionario tendrá dos llaves, "x" y "y", que tendrán asociadas los respectivos componentes de cada coordenada. Su función debe retornar un valor de verdad que indique si los tres puntos se encuentran sobre la misma recta o no.

5. Enunciado: El juego de las Picas y Fijas es un juego matemático muy sencillo, que consiste en adivinar un número de 4 cifras cuyos dígitos son todos diferentes. Para esto, el jugador que intenta adivinar deberá decir el número que cree está escondiendo el otro, y este deberá responder el número de picas y fijas que tiene ahora el jugador.

Una *pica* es un dígito que se encuentra en el número a adivinar pero no está en el lugar correcto, y una *fija* es un dígito correctamente colocado.

Por ejemplo, si el número a adivinar es [1234](#), y otro jugador dice [1325](#), tendrá dos picas y una fija.

Usted debe crear una función que devuelva un diccionario con las llaves "[PICAS](#)" y "[FIJAS](#)" que represente el resultado de la jugada si un jugador trata de adivinar el [numero_secreto](#) con el número [intento](#).

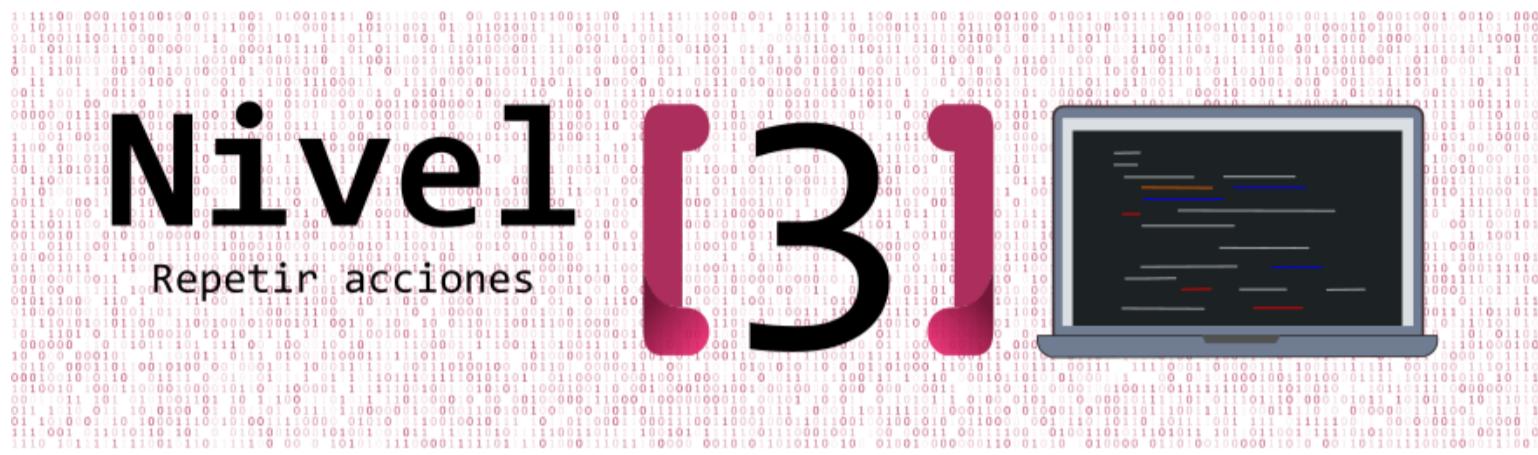
By Mario Sánchez

© Copyright Agosto de 2020.

[Departamento de Ingeniería de Sistemas y Computación](#) - Universidad de los Andes

 On this page**⚠ Versión borrador / preliminar**[Print to PDF ▶](#)

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.



4. Introducción

Éramos de provincia y ahora somos intergaláticos.

—I. Capasso

En este nivel se presentan los conceptos necesarios para que un conjunto de instrucciones se ejecute varias veces dependiendo de alguna condición o de los datos que proporcione el usuario. Aunque suena simple, esto hace que nuestros programas sean infinitamente más poderosos y que sea posible resolver problemas que antes eran imposibles.

El poder adicional que nos dan las instrucciones repetitivas (o iterativas) que estudiaremos en este nivel viene de la mano con una mayor complejidad en los programas que vamos a construir. Es por esto que en este nivel usted debe redoblar sus esfuerzos y recordar lo que dijimos en la introducción al libro: para aprender a programar, se debe practicar programando. Además, la práctica debe ser deliberada y reflexiva: resuelta ejercicios diferentes y al terminar con cada uno reflexione sobre lo que aprendió y sobre lo que se le dificultó.

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

- Instrucciones repetitivas (ciclos)
- Estructuras de datos de una dimensión (listas)
- Recorrido de secuencias (listas y cadenas de caracteres)
- Algorítmica
- Archivos de texto

By Mario Sánchez
© Copyright Agosto de 2020.

[Departamento de Ingeniería de Sistemas y Computación](#) - Universidad de los Andes

[On this page](#)

- [4.1.1. Slicing básico: inicio y fin](#)
- [4.1.2. Slicing intermedio: posiciones negativas](#)
- [4.1.3. Slicing avanzado: cambiando el paso](#)
- [4.1.4. Slices sobre listas](#)
- [4.1.5. Reemplazando slices por ciclos](#)
- [4.1.6. Modificar listas usando slices](#)
 - [4.1.6.1. Mínimo Común Múltiplo](#)
- [4.1.7. Ejercicios](#)
- [4.1.8. Más allá de Python](#)

⚠ Versión borrador / preliminar[Print to PDF ▶](#)

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.

4.1. Slicing

i Objetivo de la sección

El objetivo de esta sección es presentar la herramienta de *slicing* ofrecida por Python y demostrar su uso sobre listas y cadenas.

La expresión *slicing* hace referencia a la operación por medio de la cual se extraen elementos de una secuencia, tal como una lista o una cadena de caracteres. Dependiendo del caso, los elementos podrían ser consecutivos o podrían estar separados dentro de la secuencia original. Dado que esta operación es de uso bastante frecuente, Python ofrece características que hacen posible construir *slices* de forma breve, sin tener que construir funciones especializadas para hacerlo.

Esta sección explica cómo crear *slices* en Python a partir de listas y cadenas de caracteres [1]. Además, se presentará la forma en la cual se podrían implementar funcionalidades equivalentes utilizando ciclos.

4.1.1. Slicing básico: inicio y fin

Supongamos que tenemos la siguiente cadena de caracteres con la que inicia el himno de Colombia:

```
cadena = '¡Oh, gloria inmarcesible! ¡Oh, júbilo inmortal!'
```

En secciones anteriores ya hemos estudiado cómo podemos hacer para extraer elementos de esta cadena usando paréntesis cuadrados y una posición numerada desde cero.

```
>>> cadena[0]
'i'
>>> cadena[1]
'0'
```

Para extraer más de un elemento podemos indicar la posición inicial y la posición final usando `:` como separador. Eso lo hemos hecho en los siguientes ejemplos:

```
>>> cadena[0:11]
'¡Oh, gloria'
>>> cadena[1:11]
'Oh, gloria'
```

En el primer caso estamos extrayendo un *slice* de la cadena original que tiene sólo los caracteres desde la posición 0 hasta la 11 (excluida), mientras que en el segundo caso lo hacemos desde la posición 1 hasta la 11 (excluida). Es decir que, la posición inicial se incluye dentro del *slice* resultante, pero la posición final siempre queda por fuera.

Es también frecuente que se quiera extraer un conjunto de elementos que se encuentran al final o al principio de una cadena. En estos casos lo que se debe hacer es no incluir explícitamente el inicio o el fin, pero sí utilizar el separador `:`. Si no se quiere dejar el espacio vacío, otra alternativa es usar `None` para la posición. Esto puede verse en los siguientes ejemplos:

```
>>> cadena[30:]
'júbilo inmortal!'
>>> cadena[:30]
'¡Oh, gloria inmarcesible! ¡Oh,
>>> cadena[None:30]
'¡Oh, gloria inmarcesible! ¡Oh,'
```

En el primer caso, se extrae un *slice* con los caracteres que se encuentran a partir de la posición 30. En el segundo caso, se extrae un *slice* con los caracteres que se encuentran desde el inicio de la cadena hasta el carácter 30 (excluido).

Algo para notar es que hay una diferencia importante entre no incluir el fin e incluir un valor explícito: como la última posición nunca se incluye en el *slice*, al no incluir la posición final se está asumiendo que se debe llegar hasta el final; si por el contrario se hiciera explícita la última posición, el último valor no se incluiría en el *slice*.

```
>>> cadena[40:]
'mortal!'
>>> cadena[40:46]
'mortal'
```

Veamos ahora cómo podríamos hacer para extraer la primera frase del himno. Recuerde que el método `index` permite saber en qué posición aparece un determinado carácter dentro de una cadena.

```
>>> cadena[cadena.index('i') : cadena.index('!')+1]
'iOh, gloria inmarcesible!'
```

Finalmente, si no se incluyen ni la posición inicial ni la final, se entiende que se quiere un *slice* que incluya todos los elementos. En el caso de las listas, esto es equivalente a crear una copia de la estructura original.

```
# Crear una copia de La cadena original
cadena2 = cadena[:]
```

4.1.2. Slicing intermedio: posiciones negativas

Algo muy interesante en Python con respecto a los *slices*, es que las posiciones de inicio y de fin no necesariamente tienen que ser números positivos: si se usan números negativos, Python asume que se están referenciando las posiciones numeradas desde la última hasta la primera, empezando con -1. Esto se presenta claramente en la [figura 4.1](#), en la que hemos numerado las posiciones:

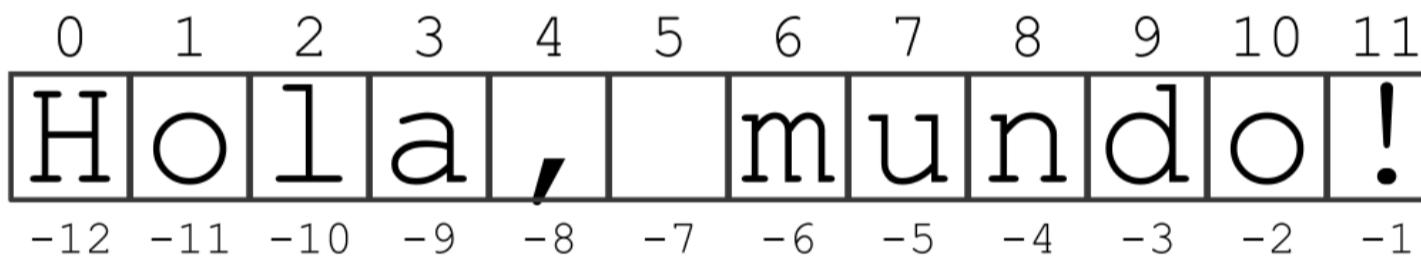


Fig. 4.1 Posiciones numeradas en una cadena

Acá podemos ver que cada carácter tiene dos números que indican la posición: uno positivo y uno negativo. Por ejemplo, la última posición de la cadena es la `11` (uno menos que el tamaño de la cadena) y es también la posición `-1`. El penúltimo carácter se encuentra en la posición `10` o, más explícitamente, en la posición `-2`. Finalmente, veamos que el primer carácter de la cadena se encuentra en la posición `0` y en la posición `-12`. Es decir que el primer carácter se puede encontrar siempre multiplicando por `-1` el tamaño de la cadena.

Veamos ahora un ejemplo en que consultamos posiciones negativas sobre nuestra cadena de ejemplo:

```
>>> cadena[-1]
'!'
>>> cadena[-2]
'1'
```

Las posiciones positivas y negativas son equivalentes y son intercambiables. En los siguientes ejemplos se pueden ver combinaciones de posiciones positivas y negativas para indicar el inicio y el fin de un *slice*.

```
>>> cadena[-47: -36]
'iOh, gloria'
>>> cadena[0: -36]
'iOh, gloria'
>>> cadena[-47: 11]
'iOh, gloria'
>>> cadena[-47:]
'iOh, gloria inmarcesible! ¡Oh, júbilo inmortal!'
>>> cadena[: -1]
'iOh, gloria inmarcesible! ¡Oh, júbilo inmortal'
```

4.1.3. Slicing avanzado: cambiando el paso

Estudiaremos ahora el tercer parámetro disponible para describir un *slice* en Python, el cual también se separa usando el carácter `:`. Este tercer parámetro, que tiene como valor por defecto `1`, indica cómo se deben recorrer las posiciones desde el inicio hasta el fin. Así, cuando su valor es `1` con cada *paso* se va a sumar `1` a la posición actual, mientras que cuando el valor es `-2` con cada *paso* se resta `2` a la posición actual.

Veamos cómo funciona esto sobre nuestra cadena de ejemplo:

```
>>> cadena[::2]
'¡h lraimreil!¡h úioimra!'
>>> cadena[::-2]
'!armioíú h¡!liermiarl h¡'
```

En el primer caso, se extrae toda la cadena con un paso de `2`: se parte de la posición `0` y se extraen las letras de la cadena, avanzando de `2` en `2` hasta llegar al final de la cadena. En el segundo caso, el paso es `-2`: se parte de la última posición y se extraen las posiciones retrocediendo de `2` en `2` hasta llegar a la cadena.

El tercer parámetro de un *slice* es particularmente útil para invertir una cadena. Así, un *slice* descrito como `[::-1]` siempre servirá para obtener una cadena invertida.

```
>>> cadena[::-1]
'!latromni olibúj ,h0¡ !elbisecramni airolg ,h0¡'
```

Al usar el parámetro del *paso*, se debe tener cuidado de construir una expresión coherente. Por ejemplo, si se quiere recorrer desde la posición `0` hasta la `5`, el paso no podría ser `-1` porque desde `0` sería imposible llegar a `5` restando `1` cada vez. Tampoco tendría sentido intentar ir de `-1` a `0` con un paso de `2`.

4.1.4. Slices sobre listas

En las secciones anteriores hemos ilustrado el uso de *slices* sólo sobre cadenas de caracteres. Sin embargo, es posible también aplicar estas mismas técnicas sobre listas, como se muestra a continuación:

```
>>> valores
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> valores[0:5]
[0, 1, 2, 3, 4]
>>> valores[5:]
[5, 6, 7, 8, 9, 10]
>>> valores[2:-1:2]
[2, 4, 6, 8]
>>> valores[-1::-2]
[10, 8, 6, 4, 2, 0]
>>> valores[-1:0:-2]
[10, 8, 6, 4, 2]
```

Aunque sencillos, estos ejemplos muestran que el mecanismo de *slicing* se aplica exactamente igual sobre listas que sobre cadenas de caracteres.

4.1.5. Reemplazando slices por ciclos

La funcionalidad de *slicing* de Python es muy útil porque simplifica el uso de operaciones que se requieren frecuentemente. Sin embargo, no muchos lenguajes ofrecen capacidades comparables para describir *slices* como en Python, así que es conveniente saber cómo reemplazar esta funcionalidad con funciones basadas en ciclos.

En el siguiente fragmento de código mostramos la función `extraer_slice`, la cual permite construir *slices* de forma prácticamente equivalente a como lo hace Python con su sintaxis especializada.

```
def extraer_slice(elementos: list, inicio: int, fin: int, paso: int) -> list:
    """ Extrae un slice de una lista.
    Parámetros:
        elementos (list): La lista de la que se van a extraer los elementos.
        inicio (int): La primera posición desde la que se van a extraer los elementos. Puede ser un número negativo.
        fin (int): La última posición de la que se van a extraer los elementos. Puede ser un número negativo.
        paso (int): indica cada cuántas posiciones se va a extraer un elemento. Puede ser un número negativo.
    Retorno:
        (list): Una lista con los elementos extraídos desde 'inicio' hasta 'fin', donde los elementos extraídos están separados por 'paso' posiciones.
    """
    tam = len(elementos)
    # Si la posición inicial es negativa, se busca el número positivo equivalente
    if inicio < 0:
        inicio = tam + inicio
    # Asegurar que el inicio siempre esté dentro de la lista
    inicio = max(0, inicio)
    inicio = min(inicio, tam-1)
    # Si la posición final es negativa, se busca el número positivo equivalente
    if fin < 0:
        fin = tam + fin

    # Preparar la lista resultante
    slice = []
    posicion = inicio
    # El ciclo se repite mientras siga siendo posible avanzar desde
    # La posición actual hacia la posición final y se estén consultando
    # posiciones que estén dentro de la lista
    while (posicion >= 0 and posicion < tam) and \
        ((posicion < fin and paso > 0) or (posicion > fin and paso < 0)):
        slice.append(elementos[posicion])
        posicion += paso
    return slice
```

Las principales diferencias entre esta función y las funcionalidades nativas de Python son las siguientes:

- Nuestra función no tiene valores por defecto, así que todos los parámetros son obligatorios.
- Se debe especificar explícitamente el fin del recorrido (no se puede dejar en blanco).

Veamos ahora cómo se comporta la función cuando se aplica a una lista de números.

```
>>> valores = [0,1,2,3,4,5,6,7,8,9,10]
>>> print(extraer_slice(valores,2,9,1))
[2, 3, 4, 5, 6, 7, 8]
>>> print(extraer_slice(valores,2,9,3))
[2, 5, 8]
>>> print(extraer_slice(valores,-9,9,3))
[2, 5, 8]
>>> print(extraer_slice(valores,9,-9,-3))
[9, 6, 3]
>>> extraer_slice(valores, 5, 50, 1)
[5, 6, 7, 8, 9, 10]
>>> extraer_slice(valores, 50, 5, -1)
[10, 9, 8, 7, 6]
```

4.1.6. Modificar listas usando slices

Además de permitir la extracción de partes de una cadena o de una lista, los *slices* también tienen un rol importante en la modificación de los elementos de una lista.

Empecemos por el caso más sencillo, en el cual se calcula un slice con sólo una posición:

```
>>> valores = [0,1,2,3,4,5,6,7,8,9,10]
>>> valores[1] = 99
>>> valores
[0, 99, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Si en cambio calculamos un *slice* que pueda tener uno o más valores, es necesario que el valor que asignemos sea también una lista para que pueda remplazar a los elementos del *slice*. En el siguiente ejemplo, primero remplazamos el *slice* de tamaño 1 que se calcula entre las posiciones 3 y 4. A continuación, calculamos un *slice* que va desde la posición 4 hasta la 7 y lo reemplazamos por la lista que sólo tiene el número 97. Finalmente reemplazamos todos los elementos que se encuentran a partir de la posición 5 por los elementos de la lista [96, 95, 94].

```
>>> valores[3:4] = [98]
>>> valores
[0, 99, 2, 98, 4, 5, 6, 7, 8, 9, 10]
>>> valores[4:7] = [97]
>>> valores
[0, 99, 2, 98, 97, 7, 8, 9, 10]
>>> valores[5:] = [96, 95, 94]
>>> valores
[0, 99, 2, 98, 97, 96, 95, 94]
```

En los ejemplos anteriores, sólo utilizamos el avance por defecto (1). Si queremos utilizar un avance diferente, positivo o negativo, tenemos que tener en cuenta que los valores que vayamos a asignar tienen que tener exactamente el mismo tamaño que el *slice* que se esté calculando. Observemos los siguientes ejemplos que ilustran este punto.

- Después de haber inicializado de nuevo la lista de valores, intentamos asignarle una lista de tamaño 1 a un *slice* de tamaño 6: Python nos advierte del error de manera muy explícita.
- En el siguiente intento, asignamos los valores `['a', 'b', 'c', 'd', 'e', 'f']` al mismo slice lo cual sí funciona y modifica la lista de acuerdo a lo que esperábamos.
- Finalmente, calculamos un *slice* empezando desde la última posición y hacemos la asignación de un nuevo conjunto de valores: `['x', 'y', 'z']`.

```
>>> valores = [0,1,2,3,4,5,6,7,8,9,10]
>>> valores[-1::-2] = [99]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: attempt to assign sequence of size 1 to extended slice of size 6
>>> valores[::2] = ['a', 'b', 'c', 'd', 'e', 'f']
>>> valores
['a', 1, 'b', 3, 'c', 5, 'd', 7, 'e', 9, 'f']
>>> valores[-1::-4] = ['x', 'y', 'z']
>>> valores
['a', 1, 'z', 3, 'c', 5, 'y', 7, 'e', 9, 'x']
```

4.1.6.1. Mínimo Común Múltiplo

Para cerrar esta sección vamos a presentar un ejercicio en el cual la modificación de listas son *slices* resulta de mucha utilidad.

El mínimo común múltiplo de dos enteros es el número menor que es múltiplo de ambos enteros. Existen varios algoritmos para encontrar este número, pero acá vamos a presentar una propuesta que basada en el uso de *slices* y en la idea de que el mínimo común múltiplo nunca puede ser superior a la multiplicación de los dos enteros.

El algoritmo que vamos a utilizar se basa en el siguiente esquema:

- Construir dos listas de valores booleanos que tenga un tamaño igual al máximo valor posible del mínimo común múltiplo. Llenar esas listas con el valor `False`.
- Poner el valor `True` en todas las posiciones de la primera lista que correspondan a un múltiplo del primer entero.
- Poner el valor `True` en todas las posiciones de la segunda lista que correspondan a un múltiplo del segundo entero.
- Buscar la posición en la cual tanto en la primera lista como en la segunda lista se encuentre el valor `True`.

La siguiente función implementa el algoritmo y utiliza *slices* para modificar fácilmente las listas.

```
def minimo_comun_multiplo(a: int, b:int) -> int:
    maximo = a * b
    multiplos_a = [False] * (maximo+1) # Crea la primera lista
    multiplos_a[a::a] = [True]*b # Marca con True los múltiplos de a
    multiplos_b = [False] * (maximo+1) # Crea la segunda lista
    multiplos_b[b::b] = [True] * a # Marca con True los múltiplos de b
    # Empieza la búsqueda del mcm
    encontre_mcm = False
    candidato_mcm = max(a,b)
    while not encontre_mcm:
        if multiplos_a[candidato_mcm] and multiplos_b[candidato_mcm]:
            encontre_mcm = True
        else:
            candidato_mcm += 1
    return candidato_mcm
```

A continuación se muestran varios llamado a la función y el resultado que se obtiene en cada caso.

```
>>> minimo_comun_multiplo(4, 6)
12
>>> minimo_comun_multiplo(4, 5)
20
>>> minimo_comun_multiplo(4, 8)
8
>>> minimo_comun_multiplo(10, 14)
70
>>> minimo_comun_multiplo(4, 9)
36
```

Para complementar este ejemplo, presentamos una segunda versión de la función en la cual se utilizan dos elementos que no se han estudiado hasta ahora: la función `zip` y el tipo de dato `tuple`. La primera parte de esta solución es idéntica a la anterior, pero utiliza otro mecanismo para buscar el mínimo común múltiplo.

1. Primero, la función `zip` se encarga de construir parejas de valores de las dos listas (el primer elemento de una lista se empareja con el primer elemento de la segunda lista, el segundo con el segundo y así sucesivamente).
2. Luego, se utiliza el método `index` del tipo `list` para buscar la posición de un elemento. En este caso, lo que se busca es una pareja en la cual los dos valores sean `True`. La posición en la que esta pareja se encuentre debería ser el valor del mínimo común múltiplo.

```
def minimo_comun_multiplo_zip(a: int, b:int) -> int:
    maximo = a * b
    multiplos_a = [False] * (maximo+1)
    multiplos_a[a::a] = [True]*b
    multiplos_b = [False] * (maximo+1)
    multiplos_b[b::b] = [True] * a
    parejas = list(zip(multiplos_a, multiplos_b))
    candidato_mcm = parejas.index((True, True))
    return candidato_mcm
```

```
>>> minimo_comun_multiplo_zip(4, 6)
12
>>> minimo_comun_multiplo_zip(10, 14)
70
>>> minimo_comun_multiplo_zip(4, 9)
36
```

Finalmente vamos a presentar una solución a este problema que es considerablemente más eficiente y que se basa en que el mínimo común múltiplo se puede calcular a partir del máximo común divisor.

$$mcm(a, b) = \frac{|a \cdot b|}{MCD(a, b)}$$

```
import math
def minimo_comun_multiplo_rapido(a: int, b:int) -> int:
    mcm = (a*b)/math.gcd(a,b)
    return mcm
```

En este caso estamos usando la función para calcular el máximo común divisor que provee Python en su módulo `math`, pero podríamos implementar nuestra propia función si fuera necesario (en otro punto de este libro se propone este problema).

4.1.7. Ejercicios

1. Escriba una función que le sirva para *reconocer* si una palabra es o no palíndrome (se lee igual al derecho y al revés). Su función sólo debe utilizar slices.
2. Escriba una función que le permita reconocer si una frase es palíndrome (se deben ignorar los espacios, signos de puntuación y mayúsculas o minúsculas). Por ejemplo, “Anita lava la tina” es una frase palíndrome. Ayuda: el módulo `string` declara el valor `ascii_letters` que podría serle de utilidad. Eliminar los caracteres que no le sirvan probablemente requerirá de un ciclo.
3. Escriba una función que utilice `slices` para extraer las palabras que están en las posiciones impares de una frase. Por ejemplo, si se aplicara la función a la primera frase de este enunciado el resultado debería ser:
`['Escriba', 'función', 'utilice', 'para', 'las', 'que', 'en', 'posiciones', 'de', 'frase']`.
4. Escriba una función que extraiga la primera palabra de una frase (hasta el primer espacio).
5. Escriba una función que extraiga la última frase de un párrafo (desde el último punto).

6. Escriba una función que calcule los números primos hasta un cierto límite utilizando el algoritmo conocido como la Criba de Eratóstenes. El principio fundamental de este algoritmo es que una vez se sabe que un número es primo, ninguno de los múltiplos de ese número va a ser un número primo así que no es necesario verificar nada sobre ellos. La modificación de listas con *slices* es particularmente útil para solucionar este problema. Ayuda: Recuerde que las listas se pueden construir usando repeticiones (ej: `[False]*50`).

4.1.8. Más allá de Python

Otros lenguajes, como Julia o Go, ofrecen capacidades similares a las que ofrece Python para hacer *slicing*. Sin embargo, al no ser una característica particularmente difundida, es importante tener la capacidad para escribir los algoritmos equivalentes.

[1] Aunque en esta sección nos concentramos en listas y cadenas de caracteres, otros tipos de datos pueden soportar *slicing* también. En particular, soportar estas operaciones requiere la implementación cuidadosa del método `__getitem__`.

By Mario Sánchez
© Copyright Agosto de 2020.

[Departamento de Ingeniería de Sistemas y Computación](#) - Universidad de los Andes

 On this page**⚠ Versión borrador / preliminar**[Print to PDF ▶](#)

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.



5. Introducción

It is much more rewarding to do more with less.

—Donald Knuth

En este nivel nos enfrentaremos a problemas un poco más complicados que los de los niveles anteriores y para eso tenemos que recurrir a herramientas más poderosas. Por una parte, estudiaremos matrices, una estructura de datos que nos permite organizar información usando una, dos o más dimensiones. Esto lo aplicaremos a casos como el procesamiento de imágenes.

Por otra parte, estudiaremos una librería para el análisis de datos y la creación de gráficas llamado Pandas. Aunque esta librería es muy útil y su uso está ampliamente difundido en la actualidad, el objetivo principal de estudiar Pandas es que usted vea que hay muchas librerías disponibles para facilitarle su trabajo, pero que también hay que desarrollar algunas habilidades para poder usar esas librerías de forma efectiva. Por ejemplo, es necesario ser capaz de buscar documentación, encontrar las partes más relevantes, leer y adaptar ejemplos y buscar ayuda en Internet.

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

- Tuplas
- Matrices
- Imágenes RGB usando Matplotlib
- Análisis de datos usando Pandas
- Gráficas usando Pandas

By Mario Sánchez
© Copyright Agosto de 2020.

[Departamento de Ingeniería de Sistemas y Computación](#) - Universidad de los Andes

[On this page](#)[5.1.1. El tipo tuple en Python](#)[5.1.2. Empaquetado y desempaquetado](#)[5.1.3. Uso de tuplas](#)[5.1.3.1. Creación de rectángulos](#)[5.1.3.2. Estadísticas de una lista](#)[5.1.4. Ejercicios](#)[5.1.5. Más allá de Python](#)

Versión borrador / preliminar

[Print to PDF ▶](#)

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.

5.1. Tuplas

Objetivo de la sección

El objetivo de esta sección es presentar el tipo de dato *tupla* y explicar cuáles son las ventajas que trae su uso en casos muy particulares.

Al inicio de este libro presentamos los tipos str, int, float y bool. Todos estos son tipos en los cuales hay un único valor que nos interese. Luego estudiamos los tipos dict y list, que sirven para representar valores múltiples con muchísima libertad: una lista puede tener cualquier cantidad de elementos, así como un diccionario puede tener cualquier cantidad de llaves. Todos estos tipos son de utilidad en muchísimos casos, pero tienen una limitación: no nos sirven en casos en los que se quiera tener un conjunto limitado de valores.

Para ilustrar el problema, tomemos el caso de una coordenada cartesiana (*x,y*). Si queremos representar esta coordenada, podríamos utilizar dos variables de tipo float, o podríamos utilizar una lista con dos números flotantes. En el primer caso, el problema es que estaríamos usando dos valores para representar una sola coordenada. Así, construir una función que retornara una coordenada sería un problema porque una función sólo puede retornar un valor. En el segundo caso, tendríamos una sola lista con los dos valores así que el problema del retorno no se presentaría. El problema acá sería que no habría ninguna garantía de que la lista retornada tenga exactamente 2 posiciones: podría tener 1, 2, 3, más de 3 o incluso podría ser una lista vacía. Aunque nosotros podríamos verificar el tamaño, sería mucho más conveniente tener un tipo de dato que nos permita agrupar múltiples valores y al mismo tiempo nos dé un poco más de garantía con respecto a la cantidad y a su orden.

En Python, esto lo vamos a lograr con el tipo de dato tuple, llamado *tupla* en español, el cual sirve para representar una secuencia inmutable de valores. A continuación estudiaremos este tipo de datos e ilustraremos su uso con varios ejemplos.

5.1.1. El tipo tuple en Python

Para ilustrar el uso del tipo tuple, usaremos el ejemplo de las coordenadas cartesianas que introducimos más arriba.

En primer lugar, veamos que la construcción de una nueva tupla requiere que se especifique desde el inicio cuáles van a ser sus valores:

```
>>> coordenada = (3, 4.5)
>>> print(type(coordenada))
<class 'tuple'>
```

Como se ve en este ejemplo, una tupla se crea usando paréntesis redondos, a diferencia de los cuadrados que se usan para las listas, y las llaves que se usan para los diccionarios. Los valores que irán dentro de la tupla se separan siempre usando comas y, al igual que en las listas, se organizan dentro de la tupla en el orden en que se especifican iniciando con la posición 0.

Si queremos consultar los valores dentro de una tupla tenemos que usar la posición al igual que como se hace en una lista:

```
>>> print(coordenada[0], "-", coordenada[1])
3 - 4.5
```

Las tuplas también se pueden recorrer exactamente igual a como se recorrería una lista:

```
>>> for valor in coordenada:
...     print(valor)
3
4.5
>>> for pos in range(len(coordenada)):
...     print(pos, "-", coordenada[pos])
...
0 - 3
1 - 4.5
```

La gran diferencia entre las tuplas y las listas es que las tuplas son inmutables. Esto quiere decir que después de que una tupla se haya creado, es imposible modificar los valores que contiene. Esto quiere decir que no existen métodos equivalentes a `append` e `insert`, y que no es posible hacer una asignación a una tupla ya creada.

```
>>> coordenada[0] = -1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

5.1.2. Empaquetado y desempaquetado

Una característica muy útil de las tuplas en Python es que permiten hacer lo que denomina *empaquetado* y *desempaquetado* de valores. *Empaquetado* hace referencia a lo que ya vimos: tomar varios valores y agruparlos en una única tupla. El *desempaquetado* es el proceso inverso de tomar los valores contenidos en una tupla y dejarlos en variables separadas. Lo que hace esto interesante es que, en lugar de requerir varias instrucciones con asignaciones, se puede hacer en una sola instrucción.

El siguiente fragmento muestra cómo se podrían desempaquetar los valores de una coordenada sin tener que acceder a cada una de las posiciones en la tupla:

```
x, y = coordenada
```

Si no se usara la capacidad de desempaquetar, el código habría tenido que ser el siguiente.

```
x = coordenada[0]
y = coordenada[1]
```

5.1.3. Uso de tuplas

Las tuplas en Python pueden utilizarse en muchas situaciones diferentes en las que también podrían utilizarse listas. Sin embargo, es recomendable que se usen cuando se requieran las siguientes características:

- Agrupar varios valores, asegurando la cantidad de valores
- Inmutabilidad

Como ya vimos antes, un uso muy común para las tuplas es servir para agrupar y organizar valores retornados por una función. En este caso, el uso de tuplas nos ayuda a garantizar que los valores retornados por la función estén siempre en el orden que se espera y que no sean modificados más adelante.

Otro uso muy común es agrupar valores que siempre deberían estar juntos. Por ejemplo, si se está trabajando con coordenadas cartesianas las tuplas servirán para agrupar los valores para cada uno de los ejes.

Veamos a continuación otros ejemplos de uso de tuplas.

5.1.3.1. Creación de rectángulos

En este caso tenemos una función que recibe las coordenadas (x,y) de dos puntos y encuentra el rectángulo delimitado por esos puntos. Para esto, la función busca la esquina del rectángulo con menor coordenadas x y y , y calcula el ancho y alto del rectángulo.

```
def crear_rectangulo(punto1: tuple, punto2: tuple) -> tuple:
    x1, y1 = punto1
    x2, y2 = punto2
    ancho = abs(x1 - x2)
    alto = abs(y1 - y2)
    x = min(x1, x2)
    y = min(y1, y2)
    return (x, y, ancho, alto)
```

A continuación vemos cómo se puede utilizar el mecanismo de desempaquetado para llamar a la función y almacenar el resultado en cuatro variables.

```
x, y, ancho, alto = crear_rectangulo(50, 180, 120, 30)
```

5.1.3.2. Estadísticas de una lista

En este caso tenemos una función que calcula 3 estadísticas sobre una lista: el menor valor, el mayor valor y el valor promedio. Esta función recibe la lista de valores y retorna una tupla con los tres números calculados. Lo interesante de este ejemplo es que normalmente se habrían necesitado 3 funciones, cada una con su propio recorrido sobre la lista, para calcular los tres valores. En este caso estamos aprovechando para calcular todo lo que necesitamos con un único recorrido.

```
def estadisticas(valores: list) -> tuple:
    mayor = valores[0]
    menor = valores[0]
    total = 0
    for valor in valores:
        total += valor
        if valor > mayor:
            mayor = valor
        elif valor < menor:
            menor = valor
    promedio = total / len(valores)
    return (menor, mayor, promedio)
```

A continuación vemos cómo se puede utilizar el mecanismo de desempaquetado para llamar a la función y almacenar el resultado en tres variables.

```
menor, mayor, promedio = estadisticas(valores)
```

5.1.4. Ejercicios

1. Construya una función que reciba dos vectores y retorne un nuevo vector que sea la suma de los dos vectores recibidos. Cada vector debe recibirse como una tupla con dos valores flotantes.
2. Construya una función que reciba una cadena de caracteres y retorne una lista de parejas (tuplas) donde la primera posición corresponda a una palabra que aparezca en la cadena y la segunda posición corresponda a la cantidad de veces que aparece la palabra en la cadena. La lista retornada tiene que tener tantas parejas como palabras *diferentes* haya en la cadena original.
3. En este ejercicio vamos a construir un rudimentario algoritmo de compresión que nos va a servir para reducir cadenas de ceros y unos. Lo que va a hacer el algoritmo es encontrar las posiciones en las que haya secuencias de uno o más unos y retornar una lista con la posición y tamaño de cada secuencia. Usted tiene que construir entonces una función llamada `comprimir` que reciba una cadena de caracteres (que podemos asumir que va a tener sólo los caracteres '`0`' o '`1`') y que retorne una tupla con dos elementos: el primer elemento, será la longitud original de la cadena; el segundo elemento será una lista de tuplas, donde cada tupla tiene primero el valor de una posición en la que había un '`1`' en la cadena original y luego tiene la cantidad de unos seguidos que había a partir de esa posición. Por ejemplo, si la cadena de entrada fuera '`101101111000110`', el resultado debería ser `(15, [(0, 1), (2, 2), (5, 4), (12, 2)])` porque la cadena original tenía 15 caracteres y porque había 4 secuencias de unos.
4. En este ejercicio usted debe construir la función llamada `descomprimir`, que debe ser capaz de tomar el resultado de la función `comprimir` y producir la cadena original. Es decir, si la función `descomprimir` recibe la tupla `(15, [(0, 1), (2, 2), (5, 4), (12, 2)])`, el resultado debería ser la cadena '`101101111000110`'.
5. Una universidad tiene información sobre la cantidad de estudiantes inscritos en cada curso y sobre los salones (edificio y código de salón) donde se deben dictar las clases. Dada esta información, el departamento de seguridad de la Universidad está interesado en saber cuántos estudiantes debería haber dentro de cada edificio en caso de que se tenga que realizar una evacuación. Usted debe crear una función llamada `contar_estudiantes` que reciba la información de los cursos y un horario y retorne la cantidad de estudiantes por

edificio. Las entradas de la función serán las siguientes: 1) un diccionario donde las llaves serán los nombres de los cursos y los valores serán también diccionarios con tres llaves: '`inscritos`' para indicar el número de estudiantes inscritos en el curso, '`horario`' para indicar el horario en el que se dicta el curso, y '`ubicacion`' para indicar el salón donde se dicta el curso. El horario de un curso se representa con una tupla que tiene en la primera posición el día de la semana ('lunes', 'martes', etc.) y en la segunda posición tiene una cadena con la franja horaria ('10:00 - 11:30', '11:30 - 13:00', etc.). La ubicación de un curso se representa también con una tupla que tiene en la primera posición el nombre de un edificio y en la segunda posición el código del salón (ambos son cadenas de caracteres). 2) El segundo parámetro de la función será el horario en el que se quieren contar los estudiantes (también se expresa con una tupla de número de día y franja horaria). Se puede asumir que las franjas horarias en las que se va a hacer la consulta son las mismas franjas horarias de los cursos. La función debe retornar una lista de tuplas donde cada tupla tendrá en la primera posición el nombre de un edificio y en la segunda posición la cantidad de estudiantes que deberían estar dentro del edificio en el horario consultado.

5.1.5. Más allá de Python

No es usual que los lenguajes de programación ofrezcan un tipo básico para manejar tuplas, así que aprender a utilizar este nuevo tipo no necesariamente será de utilidad en otros lenguajes. Sin embargo, muchas librerías de Python utilizan tuplas de forma frecuente, así que es conveniente adquirir destreza en su uso.

By Mario Sánchez
© Copyright Agosto de 2020.

[Departamento de Ingeniería de Sistemas y Computación](#) - Universidad de los Andes

[On this page](#)[5.2.1. Imágenes RGB](#)[5.2.2. Algoritmos básicos sobre imágenes](#)[5.2.2.1. Crear imágenes](#)[5.2.2.2. Convertir a escala de grises](#)[5.2.2.3. Binarizar](#)[5.2.2.4. Sharpening](#)[5.2.3. Visualización con Matplotlib](#)[5.2.3.1. Listas de Listas de Listas vs. Listas de Listas de Tuplas](#)[5.2.4. Ejercicios](#)[5.2.5. Más allá de Python: formatos de imágenes](#)[5.2.5.1. BMP - BitMap](#)[5.2.5.2. GIF - Graphics Interchange Format / PNG - Portable Network Graphics](#)[5.2.5.3. JPEG / JPG - Joint Photographic Experts Group](#)[5.2.5.4. SVG - Scalable Vector Graphics](#)

⚠ Versión borrador / preliminar

[Print to PDF ▶](#)

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.

5.2. Imágenes RGB

💡 Objetivo de la sección

El objetivo de esta sección es explicar los conceptos más básicos sobre el uso de imágenes en un computador, como un caso de estudio para el uso de matrices.

Prácticamente todos los dispositivos electrónicos programables de hoy en día incluyen funcionalidades para capturar, almacenar, transformar y analizar imágenes. Por ejemplo, los celulares inteligentes ofrecen múltiples mecanismos para tomar fotografías, transformarlas (rotar, cambiar el tamaño, recortar), hacerles retoques (ajustar los colores, agregar textos e imágenes), almacenarlas en la memoria y finalmente compartirlas con otros dispositivos. En el fondo, todas estas actividades implican la creación de matrices, su recorrido haciendo cálculos, y su modificación.

En esta sección estudiaremos cómo una imagen se puede representar como una matriz de píxeles para ser manipulada. Como parte de esto, estudiaremos algunos algoritmos importantes para hacer modificaciones sobre imágenes.

5.2.1. Imágenes RGB

Una imagen digital se representa usualmente con una matriz de píxeles, donde cada pixel es un punto de la imagen y tiene un determinado color [1]. Esto es análogo a como funcionan los monitores y pantallas, los cuales están compuestos por componentes electrónicos también llamados píxeles y que son capaces de cambiar de color para representar un punto de algún color. Cuando nosotros vemos una imagen en la pantalla, lo que estamos viendo en realidad es una grilla con puntos de colores y nuestro cerebro se encarga de convertirlo en una imagen suavizada y con algún sentido.

Por razones tanto históricas como técnicas, los píxeles en una pantalla o monitor usualmente están formados por 3 componentes que son capaces cada uno de producir un solo color con una intensidad variable: uno de estos componentes es capaz de producir rojos, otro sólo produce verdes y el tercero sólo produce azules. Como son tan pequeños y nuestros ojos no logran distinguirlos, los colores de estos tres componentes se combinan en nuestro cerebro para producir un solo color. Lo interesante de esto es que prácticamente cualquier color visible para un humano puede construirse como combinaciones de los tres colores primarios (aditivos) rojo, verde y azul.

De regreso al mundo de las imágenes digitales, los píxeles de una imagen se representan usualmente usando combinaciones de los mismos tres colores. A esto se le llama el modelo RGB y se ha llegado a un estándar en el cual la intensidad de cada uno de los tres colores se representa con un número entre 0 y 255 [2]: 0 significa que el color es tan oscuro que se ve negro mientras que 255 significa que el color está en su máxima intensidad. De esta manera un pixel negro en una imagen será un pixel en el cual los tres colores tengan intensidad 0, un pixel de color rojo puro será un pixel con intensidad 255 en el rojo e intensidad 0 en verde y azul, y un pixel blanco tendrá intensidad 255 en los tres componentes. La siguiente tabla muestra algunos colores con las intensidades de sus componentes:

Este sistema permite representar más de 16 millones de colores diferentes (2^{32}) a través de la combinación de 256 niveles de rojo, con 256 niveles de verde y 256 niveles de azul. El rango 0 - 255 podría parecer arbitrario pero tiene muchísimo sentido: usando 8 bits (1 byte) se pueden representar los números entre 0 y 255, así que el sistema RGB requiere de exactamente 3 bytes por cada pixel de una imagen.

5.2.2. Algoritmos básicos sobre imágenes

5.2.2.1. Crear imágenes

Considerando entonces que una imagen se representará naturalmente en Python como una matriz de tres números (un nivel de rojo, un nivel de verde y un componente azul), lo más natural será representar cada imagen como una matriz de tuplas. El siguiente programa muestra cómo se construiría una nueva imagen cuadrada de *ancho x ancho* píxeles, donde todos los píxeles serían negros.

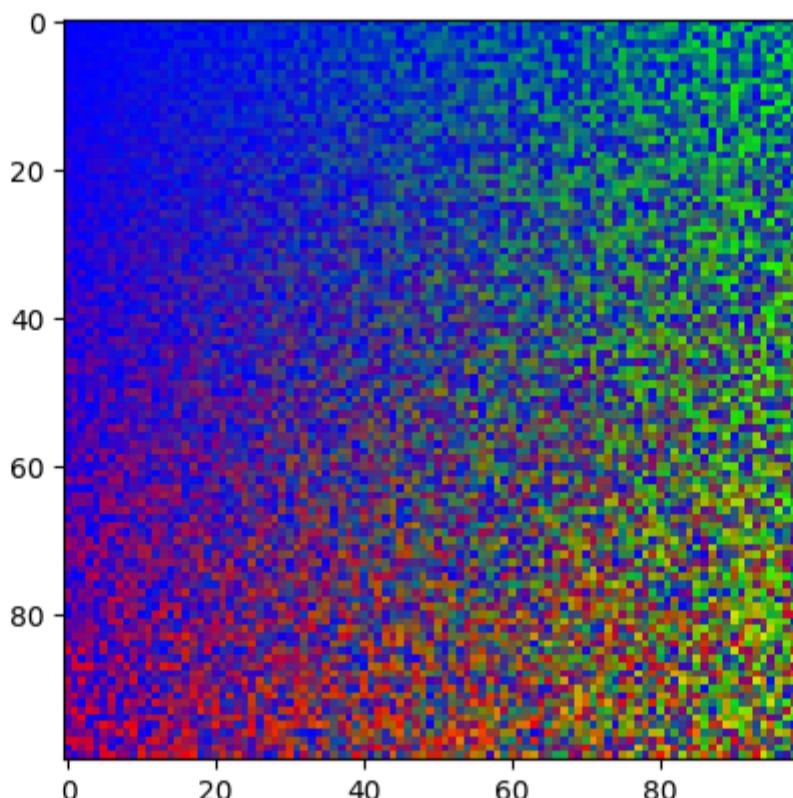
```
def generar_imagen_negra(ancho: int)->list:
    imagen = []
    for i in range(0, ancho):
        fila = []
        for j in range(0, ancho):
            rojo = 0.0
            verde = 0.0
            azul = 0.0
            pixel = (rojo, verde, azul)
            fila.append(pixel)
        imagen.append(fila)
    return imagen
```

Ahora vamos a construir una nueva función capaz de generar una imagen también cuadrada pero con pixeles de colores aleatorios.

```
import random

def generar_imagen_tuplas(ancho: int)->list:
    factor = 1/ancho
    imagen = []
    for i in range(0, ancho):
        fila = []
        for j in range(0, ancho):
            rojo = random.randint(0, i)/ancho
            verde = random.randint(0, j)/ancho
            azul = max(0,1 - rojo - verde)
            pixel = (rojo*255, verde*255, azul*255)
            fila.append(pixel)
        imagen.append(fila)
    return imagen
```

El resultado de invocar esta función una vez y de visualizar la matriz resultante en la pantalla se muestra en la siguiente imagen. El código es muy similar al de la función anterior y sólo se diferencia en la forma de generar números de forma aleatoria para el componente rojo, verde y azul de cada pixel. Más adelante en esta sección mostraremos cómo visualizar la imagen.



Para el resto de esta sección usaremos la siguiente imagen para ilustrar el efecto de cada algoritmo.



5.2.2.2. Convertir a escala de grises

Como ya vimos, en el modelo RGB el negro se forma con componentes rojos, verde y azul con valor 0, mientras que el blanco se forma con los 3 componentes con su máximo valor. Los colores intermedio (grises) tienen la propiedad de tener los tres componentes con la misma intensidad. Así, un pixel con valores `(50, 50, 50)` será un gris bastante oscuro, mientras que un pixel con valores `(200, 200, 200)` será un gris bastante claro.

Convertir una imagen a escala de grises requiere entonces recorrer la imagen entera, pixel por pixel, y modificar los componentes rojo, verde y azul para que sean iguales. La siguiente función hace esta tarea: calcula el valor promedio de los tres componentes y le asigna este valor a los tres componentes en cada uno de los píxeles. De esta manera se conserva la intensidad (qué tan iluminado es el pixel) pero se pierde el tono de todos los píxeles.

```
def convertir_a_grises(imagen: list)->None:
    alto = len(imagen)
    ancho = len(imagen[0])
    for i in range(0, alto):
        for j in range(0, ancho):
            rojo, verde, azul = imagen[i][j]
            gris = (rojo + verde + azul) // 3
            imagen[i][j] = (gris, gris, gris)
```

Al aplicar esta función a la imagen de muestra el resultado es el que se observa en la siguiente imagen.



5.2.2.3. Binarizar

Otra técnica que se aplica sobre muchas imágenes transforma los pixeles para que sean sólo negros o blancos.

Para esto se calcula el nivel de intensidad original de cada pixel y se convierte en negro o blanco según si la intensidad está por debajo o por encima de un umbral definido. Es decir, todos los pixeles cuya intensidad esté por debajo del umbral se convertirán en pixeles negros mientras que el resto se convertirán en pixeles blancos.

La siguiente función implementa el algoritmo de binarización (o umbralización) aplicado a una imagen usando un umbral que llega por parámetro.

```
def binarizar(imagen: list, umbral: int)->None:
    alto = len(imagen)
    ancho = len(imagen[0])
    for i in range(0, alto):
        for j in range(0, ancho):
            rojo, verde, azul = imagen[i][j]
            gris = (rojo + verde + azul) // 3
            if gris < umbral:
                imagen[i][j] = (0,0,0)
            else:
                imagen[i][j] = (255,255, 255)
```

Al aplicar esta función a la imagen de muestra, usando un umbral de 100, el resultado es el que se observa en la siguiente imagen.



5.2.2.4. Sharpening

El último algoritmo que vamos a presentar es una versión simplificada de un algoritmo para eliminar zonas borrosas de imágenes. Es decir, se quiere encontrar y resaltar los bordes de los objetos que aparecen en una foto.

Para lograr este efecto el algoritmo aplica una máscara a cada uno de los pixeles de tal forma que el color de cada pixel termina dependiendo de los colores de los pixeles vecinos. En nuestro caso, la máscara que vamos a utilizar es la siguiente:

-1 -1 -1

-1 9 -1

-1 -1 -1

Para aplicar la máscara en un determinado pixel lo que se tiene que hacer es centrar la máscara sobre el pixel y multiplicar los colores de los pixeles vecinos por el valor que le corresponde en la máscara. Así, en este caso el pixel que se encuentra arriba y a la izquierda del pixel que nos interesa se multiplicará por -1. Al final, se sumarán los 9 resultados de las 9 multiplicaciones y ese valor es el que se asignará al pixel en el que se centró la máscara.

Algo muy importante de este algoritmo es que el nuevo color de cada pixel debe calcularse usando los colores de los pixeles vecinos antes de que se haya aplicado el algoritmo. Esto hace necesario que, antes de empezar, se tenga que crear una copia de la imagen sobre la cual se trabajará.

```
def copiar_imagen(imagen: list) -> list:
    """ Esta función crea una copia de una imagen y la retorna
    Parámetros:
        imagen (list): es una lista de listas de tuplas que representa una imagen.
    Retorno:
        (list): retorna una copia de la imagen
    """
    copia = []
    alto = len(imagen)
    for i in range(0, alto):
        fila = imagen[i]
        nueva_fila = fila.copy()
        copia.append(nueva_fila)
    return copia

def sharpening(imagen: list) -> list:
    """ Aplica la máscara sobre la imagen. La imagen que se
    recibe como parámetro no se modifica durante el proceso.
    Parámetros:
        imagen (list): La imagen sobre la que se aplicará el algoritmo
    Retorno:
        (list): La imagen con el resultado de aplicar el algoritmo
    """

# Acá se crea la máscara que se va a aplicar
mascara = [[-1, -1, -1], [-1, 9, -1], [-1, -1, -1]]

# Crear una copia de la imagen que finalmente será retornada
copia = copiar_imagen(imagen)
alto = len(imagen)
ancho = len(imagen[0])

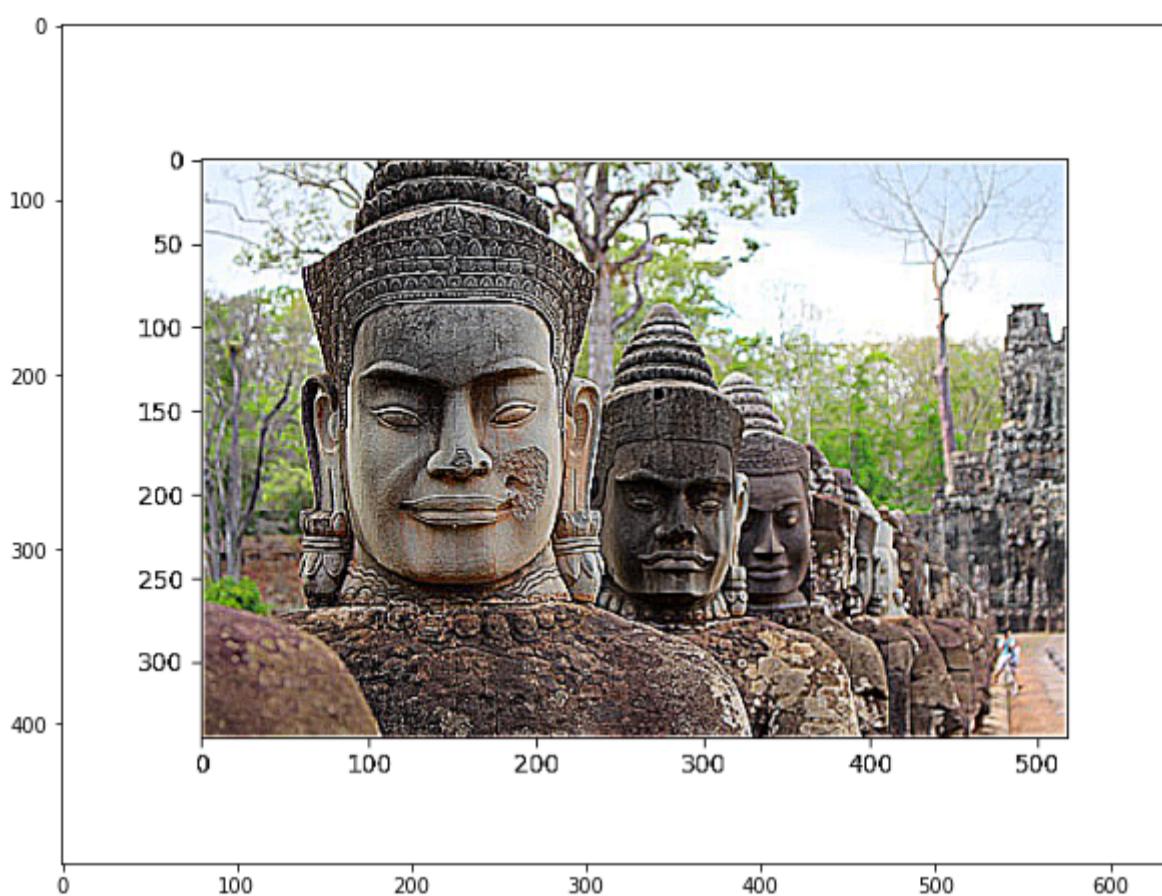
# Recorrer las filas (i) y las columnas (j) de la imagen original
# No se recorren las filas y columnas que se encuentran en el borde de
# La imagen porque no tienen vecinos completos para aplicar la máscara
for i in range(1, alto-1):
    for j in range(1, ancho-1):

        # A partir de este punto se va a aplicar la máscara al pixel que
        # se encuentra en la fila i, columnas j.
        # La máscara es de 3x3: se hacen dos ciclos para recorrer todos sus elementos.
        rojo, verde, azul = (0,0,0)
        for i_mascara in range(-1, 2):
            for j_mascara in range(-1, 2):
                # Se consultan los colores originales del pixel vecino
                rojo_vecino, verde_vecino, azul_vecino = imagen[i+i_mascara][j+j_mascara]
                valor_mascara = mascara[i_mascara+1][j_mascara+1]

                # Los colores originales se multiplican por el valor de la máscara
                # y se van sumando para encontrar el nuevo valor del color para
                # el pixel [i][j]
                rojo += rojo_vecino * valor_mascara
                verde += verde_vecino * valor_mascara
                azul += azul_vecino * valor_mascara

        nuevo_pixel = (max(rojo, 0.0), max(verde, 0.0), max(azul, 0.0))
        copia[i][j] = nuevo_pixel
return copia
```

El resultado de aplicar la función anterior a la imagen de muestra se puede apreciar en la siguiente figura. Como dijimos antes, esta es una simplificación del algoritmo para corregir fotos borrosas, así que el resultado no es muy impresionante, pero a partir de esta función usted puede intentar refinar y mejorar los algoritmos para lograr resultados realmente útiles. Por ejemplo, usted puede probar a implementar el mismo algoritmo utilizando una máscara diferente para observar cómo cambia el resultado.



5.2.3. Visualización con Matplotlib

A continuación explicaremos un mecanismo que nos permite visualizar las imágenes con las cuales hemos estado trabajando. El mecanismo está basado en el uso de la librería Matplotlib, la cual se encuentra disponible para su descarga gratuita en casi cualquier plataforma. Esta librería se ha convertido en un estándar para el manejo y creación de gráficas usando Python, así que es conveniente familiarizarse con su uso.

Matplotlib ofrece como ventaja adicional ser capaz de interpretar diferentes formatos de imágenes. Es decir que Matplotlib nos ofrece las funcionalidades necesarias para poder cargar imágenes en formato jpg o png (entre otros), sin que nosotros tengamos que preocuparnos por estos aspectos.

El siguiente programa muestra cómo se puede utilizar esta librería para cargar una imagen desde un archivo y para visualizarla dentro de un programa. Si se ejecuta este programa, y hay una imagen llamada "muestra.jpg" en la carpeta donde se está ejecutando, el resultado debería ser que se despliegue la imagen en una nueva ventana.

```
import matplotlib.image as mpimg
import matplotlib.pyplot as plt

def cargar_imagen_matriz(ruta_imagen: str) -> list:
    """ Carga la imagen que se encuentra en la ruta dada.
    Parámetros:
        ruta_imagen (str) Ruta donde se encuentra la imagen a cargar.
    Retorno:
        List: Matriz de MxNx3
    """
    imagen = mpimg.imread(ruta_imagen)
    return imagen

def visualizar_imagen_matriz(imagen: list) -> None:
    """ Muestra la imagen recibida
    Parámetros:
        imagen (List): Matriz de MxNx3 que representa la imagen a visualizar.
    """
    plt.imshow(imagen)
    plt.show()

imagen = cargar_imagen_matriz("muestra.jpg")
visualizar_imagen_matriz(imagen)
```

Algo muy importante que se tiene que tener en cuenta cuando se trabaje con Matplotlib es que la librería espera que los colores se representen con números entre 0 y 255 si son enteros, o entre 0 y 1 si son flotantes. Esto puede ser un poco confuso al principio, así que lo más conveniente es siempre usar enteros o siempre usar flotantes y no combinarlos nunca.

5.2.3.1. Listas de Listas de Listas vs. Listas de Listas de Tuplas

Las dos funciones anteriores, aunque son muy sencillas tienen un problema que las hace incompatibles con los algoritmos que hemos trabajado en esta sección: en lugar de representar las imágenes como matrices de tuplas, representan las imágenes con matrices de 3 dimensiones, donde la tercera dimensión es de tamaño 3 y permite representar cada uno de los tres componentes de una imagen. Esto quiere decir que en la posición `[i][j]` de la imagen que retorna la función `cargar_imagen_matriz` no hay una tupla sino hay una lista con tres valores.

Las siguientes funciones son versiones modificadas de las funciones anteriores que se encargan de convertir entre la representación con tuplas y la representación de Matplotlib, de forma que se pueda usar la librería para cargar los archivos y para visualizar las imágenes [3].

```
def visualizar_imagen(imagen: list) -> None:
    """ Muestra la imagen recibida
    Parámetros:
        imagen (List): Matriz de MxN con tuplas (R,G,B) que representan la imagen a visualizar.
    """
    alto = len(imagen)
    ancho = len(imagen[0])

    # Construir una matriz para representar la imagen.
    # Esta matriz tendrá tres dimensiones
    matriz = []
    for i in range(alto):
        fila = []
        for j in range(ancho):
            # convertir la tupla a una lista
            r, g, b = imagen[i][j]
            fila.append([r, g, b])
        matriz.append(fila)
    plt.imshow(matriz)
    plt.show()

def cargar_imagen(ruta_imagen: str)-> list:
    """ Carga la imagen que se encuentra en la ruta dada.
    Parámetros:
        ruta_imagen (str) Ruta donde se encuentra la imagen a cargar.
    Retorno:
        List: Matriz de MxN con tuplas (R,G,B).
    """
    matriz = mpimg.imread(ruta_imagen).tolist()
    alto = len(matriz)
    ancho = len(matriz[0])

    # Construir una matriz para representar la imagen.
    # Esta matriz tendrá dos dimensiones y tuplas en cada casilla.
    imagen = []
    for i in range(alto):
        fila = []
        for j in range(ancho):
            # Extraer los componentes. Note que no se puede desempaquetar.
            r = matriz[i][j][0]
            g = matriz[i][j][1]
            b = matriz[i][j][2]

            # Construir la tupla equivalente y agregarla a la imagen
            tupla = (r, g, b)
            fila.append(tupla)
        imagen.append(fila)
    return imagen
```

5.2.4. Ejercicios

- Escriba una función que tome una imagen y calcule su negativo. El negativo se calcula invirtiendo los 3 componentes de cada pixel de tal forma que su nuevo valor sea igual a 255 menor el valor original.
- Escriba una función que reciba una imagen y una máscara y haga la convolución entre la imagen y la máscara. Una convolución se hace igual que como se aplicó el algoritmo de “Sharpening” con las siguientes diferencias:
 - La máscara puede tener cualquier tamaño mientras sea cuadrada
 - Los valores de la máscara pueden sumar cualquier valor. En caso de que la suma sea diferente a 1, se debe dividir el valor de cada componente por el valor total de la máscara (si es positivo).
- Escriba una función que reciba una imagen y retorne tres imágenes en escala de grises que representen cada una un canal diferente (rojo, verde y azul). Esto quiere decir que en la imagen del canal rojo la intensidad de los pixeles deberá corresponder a la intensidad del color rojo en los pixeles de la imagen original. Algo similar debe ocurrir con las imágenes de los canales verde y azul.

2. Escriba una función que reciba tres imágenes en escala de grises correspondientes a los tres canales de una imagen y produzca una imagen combinada. Este mecanismo fue utilizado por el fotógrafo [Sergey Prokudin-Gorsky](#) a finales del siglo XIX para producir fotografías en color a partir de fotografías en blanco y negro tomadas con filtros de colores.
 3. Escriba una función que reciba una imagen y una lista ordenada de umbrales y produzca una imagen donde los pixeles se hayan clasificado de acuerdo al umbral donde pertenezcan. Por ejemplo, si la lista de umbrales tiene los valores [50, 100, 150], entonces:
 - todo pixel con intensidad menor a 50 se volverá negro
 - todo pixel con intensidad mayor a 150 se volverá blanco
 - todo pixel con intensidad entre 50 y 100 se volverá un gris con intensidad 75
 - todo pixel con intensidad entre 100 y 150 se volverá un gris con intensidad 125
1. Escriba una función que rote a la derecha una imagen cuadrada. Para probar, utilice una imagen con la que sea fácil distinguir que la rotación es la correcta.
 2. Escriba una función que rote a la izquierda una imagen cuadrada. Para probar, utilice una imagen con la que sea fácil distinguir que la rotación es la correcta.
 3. Escriba una función que refleje horizontalmente una imagen cuadrada. Para probar, utilice una imagen con la que sea fácil distinguir que la rotación es la correcta.
 4. Escriba una función que rote a la derecha una imagen rectangular. Para probar, utilice una imagen con la que sea fácil distinguir que la rotación es la correcta.
 5. Escriba una función que rote a la izquierda una imagen rectangular. Para probar, utilice una imagen con la que sea fácil distinguir que la rotación es la correcta.

5.2.5. Más allá de Python: formatos de imágenes

Un aspecto importante del manejo de imágenes que no se tocó en esta sección tiene que ver con los formatos utilizados para almacenar las imágenes. Cada formato ofrece diferentes ventajas y desventajas con respecto a aspectos como el nivel de compresión, el tamaño de las imágenes, la cantidad de cálculos necesarios para abrir o guardar un archivo, y la cantidad de información que se pierde debido a la compresión.

A continuación presentamos algunos de los formatos más populares.

5.2.5.1. BMP - BitMap

Este es uno de los formatos más simples puesto que almacena la información sin ningún tipo de compresión y utilizando una estructura muy similar a la que hemos estado discutiendo. Es decir, dentro de un archivo bmp se almacena la información del componente rojo, del componente verde y del componente azul de cada uno de los pixeles que forman la imagen. Esto hace que leer y escribir un archivo bmp sea muy fácil y rápido, pero lleva a archivos que pueden ser muy grandes. Por ejemplo, una imagen totalmente blanca de 320x240 pixeles ocupa en el disco aproximadamente 225kb debido a que cada uno de los 76800 pixeles requiere 3 bytes para almacenar su color.

5.2.5.2. GIF - Graphics Interchange Format / PNG - Portable Network Graphics

Los formatos GIF y PNG son conceptualmente muy similares entre ellos y se basan en la idea de incluir una paleta de colores en el archivo de la imagen, para luego no tener que describir cada color. Usualmente la paleta de colores contiene hasta 256 colores, así que cada pixel se puede representar usando sólo 1 byte en lugar de los tres bytes que utiliza bmp. Adicionalmente, GIF y PNG utilizan un algoritmo de compresión *sin pérdida* que logra disminuir el espacio utilizado, garantizando al mismo tiempo que no se pierda información. Gracias a estas características la misma imagen blanca de 320x240 pixeles que ocupaba 225kb como un bmp, ocupa 1.5kb como png y 426b como gif.

A pesar de todo esto, los dos formatos tienen una importante limitación: debido al uso de la paleta de colores, una imagen almacenada en estos formatos no debería tener una altísima variedad de colores. Esto hace que sea poco eficiente almacenar fotografías utilizando este formato. En el peor de los casos, se podría perder información porque, para reducir el tamaño de la paleta, colores similares podrían agruparse dando origen a defectos y zonas de baja calidad en las imágenes. Por este motivo, GIF y PNG son buenos formatos para almacenar especialmente imágenes creadas digitalmente y con una limitada gama de colores.

5.2.5.3. JPEG / JPG - Joint Photographic Experts Group

A diferencia de los anteriores, el formato JPG utiliza un algoritmo de compresión en el que sí se pierde información, pero de forma prácticamente imperceptible para un usuario. Como no tiene una paleta, JPG puede utilizar todos los colores que sean necesarios, por ejemplo en una foto. Donde se pierde información es en la agrupación de pixeles cercanos que tengan colores muy similares, los cuales se combinan para ganar espacio.

JPG es el formato más utilizado en el mundo para almacenar fotografías puesto que ofrece un buen compromiso entre la calidad de las imágenes y la cantidad de espacio que requiere su almacenamiento. Además, de ser necesario es posible ajustar el nivel de compresión para incrementar la calidad de las imágenes a costa de la cantidad de espacio utilizado.

Para el caso de nuestra imagen blanca de 320x240 pixeles, el archivo jpg ocupa 2kb (aproximadamente lo mismo que la imagen png). Por otro lado, en el caso de una fotografía de 5184x3456 pixeles (más de 17 millones de pixeles), el archivo png ocupa 21 megas mientras que el archivo jpg ocupa apenas 4.9 megas, sin que haya diferencias evidentes en la calidad de la imagen.

5.2.5.4. SVG - Scalable Vector Graphics

Cerramos la sección presentando brevemente el formato SVG, el cual es completamente diferente a los anteriores: mientras en BMP, JPG, PNG o GIF se intenta describir el color de cada pixel de una imagen, en SVG se describe la manera en la que debería construirse la imagen.

Lo que se presenta a continuación es la parte principal del archivo svg que describe una imagen de 320x240 pixeles en la que sólo aparece un rectángulo blanco:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<svg
  xmlns:dc="http://purl.org/dc/elements/1.1/"
  xmlns:cc="http://creativecommons.org/ns#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:svg="http://www.w3.org/2000/svg"
  xmlns="http://www.w3.org/2000/svg"
  xmlns:sodipodi="http://sodipodi.sourceforge.net/DTD/sodipodi-0.dtd"
  xmlns:inkscape="http://www.inkscape.org/namespaces/inkscape"
  width="320mm"
  height="240mm"
  viewBox="0 0 320 240"
  version="1.1"
  id="svg8"
  inkscape:version="0.92.4 (5da689c313, 2019-01-14)"
  sodipodi:docname="blanco.svg">
  <g
    inkscape:label="Layer 1"
    inkscape:groupmode="layer"
    id="layer1"
    transform="translate(0,0)">
    <rect
      style="opacity:1;fill:#ffffff;fill-opacity:1;stroke:none;stroke-width:0.30000001;stroke-
      linejoin:round;stroke-miterlimit:4;stroke-dasharray:none;stroke-dashoffset:0;stroke-opacity:1"
      id="rect815"
      width="320"
      height="240"
      x="0"
      y="0" />
  </g>
</svg>
```

La parte importante de este código es en realidad el elemento `rect`, en el cual se describe el rectángulo que se va a visualizar. El archivo resultante es de 1.7kb y tiene la ventaja de que tendría exactamente el mismo tamaño si la imagen fuera 10 veces más grande. La desventaja principal de un archivo SVG es que sólo puede utilizarse para imágenes que se puedan describir como combinaciones de figuras geométricas. No pueden utilizarse entonces para cosas como fotografía.

[1] Esta es una simplificación de la realidad, pero es suficiente para los propósitos de esta sección. Existen también imágenes en las cuales no se representan todos los pixeles sino grupos de ellos (ej. quadtrees), e imágenes que están basadas en la descripción de lo que se debería ver en ellas (ej. archivos svg).

[2] Representar colores usando combinaciones de rojo, verde y azul no es el único mecanismo utilizado en la práctica. Por ejemplo, es usual que también se tenga un componente que indica el nivel de transparencia de un pixel (alpha). La cantidad de niveles de intensidad también puede variar: si bien en muchos casos se utilizan 256 niveles para que cada pixel pueda representarse con sólo 24 bits, también hay modelos de color en los cuales se utilizan muchos más bits para representar cada componente.

[3]

En esta sección podríamos haber hecho toda la presentación del tema usando la representación preferida por Matplotlib. Sin embargo, nos parece que es conceptualmente más elegante tener una matriz de tuplas en lugar de una matriz de 3 dimensiones. En primer lugar, al usar tuplas (que son inmutables) nos aseguramos que siempre tenemos 3 valores (rojo, verde y azul). En segundo lugar, es más natural pensar en una imagen como una matriz de 2 dimensiones que como una matriz de 3 dimensiones. Finalmente, al usar tuplas podemos desempaquetar los valores, lo cual hace el código un poco más fácil de leer. Sabemos que el uso de tuplas es ligeramente más ineficiente, pero en este caso preferimos sacrificar un poco el desempeño.

By Mario Sánchez
© Copyright Agosto de 2020.

[Departamento de Ingeniería de Sistemas y Computación](#) - Universidad de los Andes