Python en Ámbitos Científicos

Facundo Batista & Manuel Carlevaro

28 de mayo de 2021

VERSIÓN PRELIMINAR

Título: Python en Ámbitos Científicos
Autores: Facundo Batista & Manuel Carlevaro

ISBN-13 (versión electrónica): ???-?-????-????-?

Primera Edición (versión preliminar)
Escrito con XeLaTeX.

Lugar: Olivos y La Plata, Buenos Aires, Argentina
Año: 2021
Web: https://github.com/facundobatista/libro-pyciencia

10 9 8 7 6 5 4 3 2 1

VERSIÓN PRELIMINAR

Prefacio

Cuando la Comunidad Científica notó que las computadoras podían facilitar su trabajo, sus integrantes se convirtieron posiblemente en los adoptantes tempranos más entusiastas de la tecnología. Aunque el uso y la potencia de las computadoras crece continuamente en el ámbito científico, poco se ha avanzado en la formación y desarrollo de habilidades en Desarrollo de Software.

El cálculo científico requiere realizar combinaciones de múltiples tareas de diversa clase. Por ejemplo, es necesario registrar automáticamente datos de un experimento y visualizarlos, realizar cálculos numéricos o simbólicos, ordenar, clasificar, simular, etc. Muchas veces podemos utilizar paquetes de software que realizan esas tareas por nosotros, pero muchas otras ocurre que nadie ha implementado un determinado cómputo en la forma que necesitamos, o simplemente queremos probar ideas nuevas. Cualquiera sea la naturaleza de nuestra actividad en la ciencia o en la tecnología, no resulta infrecuente la necesidad de interactuar con computadoras a través de programas propios.

El uso de Python en aplicaciones científicas ha aumentado sostenidamente en los últimos años, sin embargo es difícil encontrar libros o manuales en castellano de Python que no estén pensados para programadores.

Este libro nace con la idea primaria de acercar Python al mundo científico, en un libro pensado para científiques, a partir de nuestra experiencia en el dictado del curso "Herramientas Computacionales para Científicos"que ofrecemos en la Universidad Nacional de La Plata y la Universidad Tecnológica Nacional, desde 2007. De la misma manera, la elección del castellano como idioma de escritura es un factor crítico, porque aunque sabemos que el inglés es una herramienta fundamental tanto para programar como para hacer ciencia, estamos convencidos que no debería ser una barrera de entrada.

Más allá de las secciones básicas de un libro (índice, bibliografía, etc.), el libro tiene dos grandes partes. La primera habla de Python, algunas bibliotecas importantes y otros temas que son fundamentales. En la segunda, mostramos cómo abordar temas científicos básicos utilizando Python, de forma teórica y práctica.

En ambos casos este libro esquiva la pretensión de ser una referencia absoluta, sino que tiene el propósito de allanar el camino de les científiques para dar los primeros pasos en el lenguaje y solucionar los problemas básicos (pero no por eso menos importantes) de la ciencia y la tecnología.

Olivos y La Plata, Buenos Aires, Argentina,

Facundo Batista & Manuel Carlevaro

Índice general

VERSIÓN PRELIMINAR

Parte I
Herramientas fundamentales

Los capítulos siguientes desarrollan un conjunto de conocimientos y técnicas fundamentales que serán utilizados en la Parte II al abordar temas de aplicaciones específicas. Se sugiere, a los lectores que no tienen experiencia previa en el uso de Python, un recorrido secuencial a través de los capítulos que componen esta Parte.

1 | Introducción a Python

En este capítulo mostraremos las bases de Python.

Haremos un recorrido por las distintas características del lenguaje, y luego mostraremos cómo ejecutar programas y usar el intérprete en modo interactivo. Luego haremos un repaso de los tipos de datos más usados, los controles de flujo que posee el programa (incluidas las excepciones, para manejo de error), y las distintas formas de encapsular código. Finalmente indicaremos las mejores formas de pedir ayuda.

Este contenido es imprescindible para entender lo suficiente de Python como para poder leer el resto del libro. Por supuesto, no es todo lo que se puede aprender de Python, solamente son las estructuras iniciales que permitirá arrancar con el lenguaje, utilizarlo para los primeros programas, y de allí escalar todo lo que se desee.

1.1. ¿Qué es Python?

Python es un lenguaje de programación de muy alto nivel y multiparadigma.

Es un lenguaje maduro, ya que fue creado en diciembre de 1989 y usado por muchos años en todos los ámbitos posibles (desde electrodomésticos hasta en el espacio). Al mismo tiempo, es un lenguaje que no se quedó estancado, siempre está en constante evolución a través de mejoras permanentes que realiza la comunidad.

Porque es la comunidad la responsable del desarrollo y progreso del lenguaje, como así de sus herramientas relacionadas, conferencias y eventos alrededor del lenguaje. Hay grupos de desarrolladores y usuaries de Python alrededor de todo el mundo, pero el principal medio de comunicación y coordinación es online.

La comunidad es uno de los puntos fuertes de Python, ya que no sólo ofrece la capacidad técnica de mantener y evolucionar el lenguaje, o administrativa de realizar eventos, sino que provee una estructura social que cobija a les desarrolladores recién llegades lenguaje, pero no sólo al principio, sino que es una red que facilita el aprendizaje y el perfeccionamiento a lo largo de toda la vida.

¿Qué es una P.E.P.?

Es una Propuesta de Mejora de Python (Python Enhancement Proposal, en inglés), un documento que reúne las razones, ideas, discusiones de la comunidad, conclusiones, y todos los detalles relevantes con respecto a cambios grandes en el lenguaje, la biblioteca estándar, o el proyecto en general. El punto de entrada para explorarlas es la PEP 0 [2].

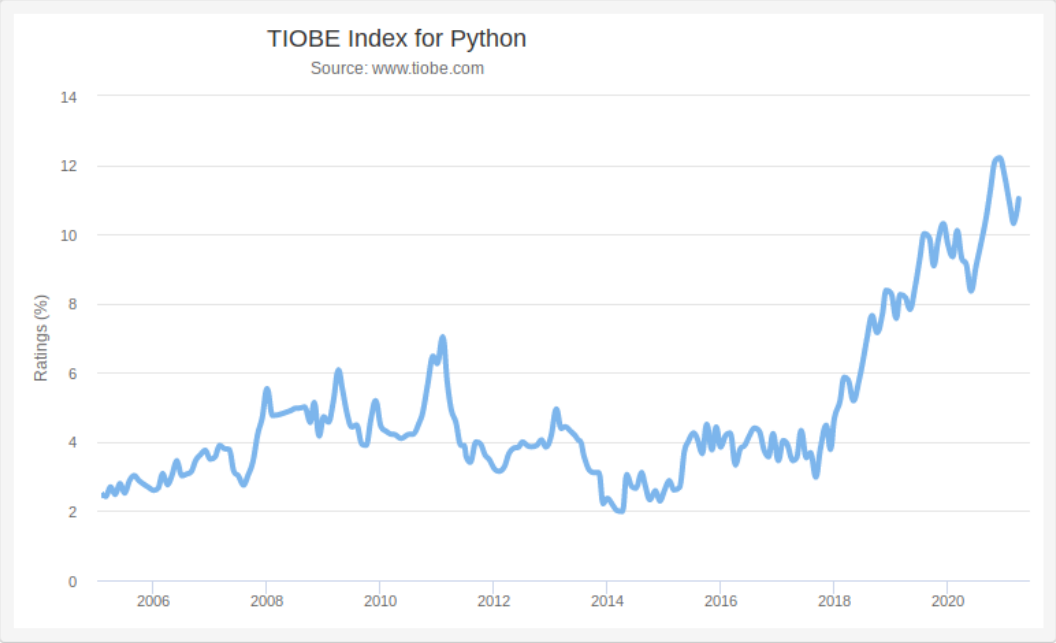


FIGURA 1.1: Evolución de la adopción de Python en el mundo

Esta misma comunidad es la que garantiza soporte y ayuda, ya que Python no depende de una empresa y no corre el riesgo de estancarse porque no es más prioridad para esa empresa (o directamente no poder usarlo más en caso de conflictos legales por copyright). A este respecto cabe mencionar a la Python Software Foundation [3], una organización sin fines de lucro encargada de proteger el copyright de Python, garantizando su libertad y disponibilidad.

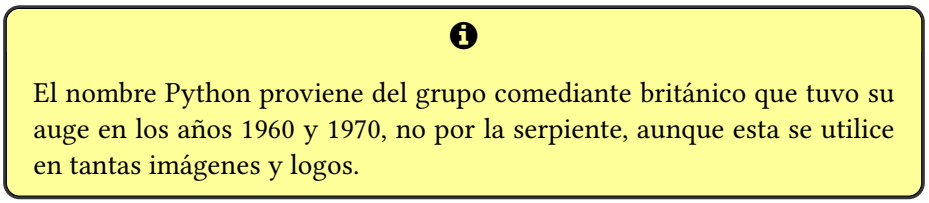
Volviendo a Python, es un lenguaje muy fácil de aprender, ya que por diseño posee una sintaxis muy sencilla, y todo el lenguaje está basado en estructuras conceptuales que se repiten todo el tiempo, bajando muchísimo la curva de adopción.

En parte por esto es que está tan recomendado para usarlo para enseñar informática, porque todo el impacto pedagógico se puede enfocar en algoritmos y lo que se quiera transmitir, sin tener que aprender primero el lenguaje en sí.

Lo mismo vemos reflejado en el mundo de la Ciencia, que fue adoptando Python en los últimos años de forma vertiginosa, ya que es un lenguaje que permite enfocarse en la ciencia a resolver y no es una traba en sí mismo.

Esta simplicidad se transmite en la cultura de Python, en conceptos como el de "debería haber una, y preferiblemente solo una, manera obvia de hacerlo". Esta y otras frases que forman el núcleo de la filosofía de Python son parte del Zen de Python [4] (que incluimos traducido en el apéndice A), una serie de indicaciones generales para pensar sobre nuestros desarrollos y para enfocar el intercambio de ideas alrededor de un pedazo de código, no reglas duras que haya que cumplir a rajatabla.

El término "pitónico" (o pythónico, o pythonic en inglés) es un neologismo común en la comunidad de Python, que expresa varios conceptos aplicados al estilo del programa. Decir que un código es pitónico es decir que usa correctamente los idiomas de Python, que se adapta a la filosofía minimalista de Python, y hace énfasis en su legibilidad, entre otros aspectos. Por el contrario, si el código es difícil de entender, trabado, o aplica idiomas traídos de otros lenguajes (con estructuras que normalmente se pueden resolver de forma más sencilla en Python), se lo denomina "no pitónico" (unpythonic).



El nombre Python proviene del grupo comediante británico que tuvo su auge en los años 1960 y 1970, no por la serpiente, aunque esta se utilice en tantas imágenes y logos.

Por último, y no menos importante, es gratis, libre, y de código abierto. Tiene una licencia muy relajada, por lo cual podemos usar Python de forma segura tanto en ámbitos estatales como privados, sin necesitar consultar con ningún abogado primero :). La única restricción que tiene a este respecto es si quisiéramos distribuir Python nosotros mismos, nada más.

1.1.1. Propiedades del lenguaje

Python es al mismo tiempo de tipado dinámico y de tipado fuerte. Dinámico, porque no hace falta declarar los nombres antes de utilizarlos. Fuerte, porque se respeta el tipo de los objetos y no se realizan conversiones implícitas. Habiendo dicho eso, tenemos que tener en cuenta que las definiciones más estrictas sobre *tipado* son sobre lenguajes que utilizan "variables", y Python no posee variables como tales, sino que son todos objetos y nombres para referenciar esos objetos (más detalle sobre este tema en la sección 1.3.5).

Posee una administración dinámica de la memoria, usando una combinación de conteo de referencias y *garbage collector* ("recolector de basura", en castellano, pero se acostumbra mencionarlo en inglés) para referencias cíclicas. Por lo tanto, aunque hay mecanismos para tener mayor o menor control de lo que sucede en memoria, en general no nos tenemos que preocupar de la misma.

Cómo mencionamos en la introducción, Python es multiparadigma. Esto implica que aunque en Python todos son objetos, nosotros podemos programar completamente de forma estructurada, e incluso utilizar muchas características y módulos que vienen de la programación funcional u orientada a aspectos. Resaltamos esta versatilidad en su aspecto pedagógico, porque normalmente se aprende programación estructurada primero y el pasaje a la programación orientada a objetos no es para nada trivial, entonces poder utilizar un lenguaje que permite empezar estructurado, e incluir eventualmente la creación de clases propias, mientras estamos expuestos a objetos todo el tiempo, hace que la curva de aprendizaje de la programación orientada a objetos sea gradual.

También en la introducción mencionamos que es un lenguaje de alto nivel. Esto es porque posee muchas estructuras de alto nivel y modernas. Resaltamos los conjuntos y diccionarios como tipos de datos integrados, funciones y clases como ciudadanos de primer orden (o sea, son simplemente objetos), módulos y paquetes para estructurar código, iteradores y generadores muy bien integrados al lenguaje, etc. Y posee un manejo moderno de errores a través de excepciones.

Aunque solemos decir simplemente que Python es un lenguaje interpretado, realmente tiene un paso de compilación interno. En detalle, la ejecución de un programa en Python se realiza en dos pasos a partir del código fuente que le pasamos: primero Python compila ese código a una series de instrucciones de la máquina virtual de Python, y luego procede a ejecutar esas instrucciones (que conmunmente denominamos *bytecode*. Esto es similar a otros lenguajes (como Java, VisualBasic, o COBOL, por ejemplo) con la diferencia fundamental que el proceso es automático: no hay intervención humana entre el primer paso de compilación y el segundo de ejecución en la máquina virtual.

Un factor clave en la velocidad de desarrollo es el intérprete interactivo, una herramienta que nos permite ejecutar pequeñas muestras de código sin tener que incluirlas en un programa propiamente dicho, facilitando al mismo tiempo una exploración de los objetos con los que estemos interactuando. Realizando esta exploración y experimentación en el intérprete interactivo,

de forma rápida y sencilla, se logra incorporar al código esos algoritmos o líneas de código ya probadas y estables, reduciendo notablemente el ciclo de prueba y error.

Las distintas funcionalidades y herramientas de Python no están integradas todas en su núcleo, sólo algunas forman parte de Python en sí, y el resto están disponibles a través de módulos y paquetes. Una gran cantidad de módulos están disponibles siempre que tengamos Python instalado en el sistema: son aquellos que forman parte de la llamada Biblioteca Estándar. Esta biblioteca es tan variada y extensa que generó la frase "Python viene con las pilas incluidas", ya que gran parte del desarrollo día a día se puede realizar sin echar mano a módulos externos.

Y también es portable, ya que corre en muchas plataformas. Y no estamos hablando solamente de las tres principales (Linux, Windows y MacOS) sino también las derivadas de Unix (como FreeBSD, o Solaris) y algunas más inusuales como Cygwin o AIX.

1.1.2. Biblioteca estándar y módulos externos

Cómo decíamos arriba, parte de la funcionalidad está integrada en Python como lenguaje, y luego tenemos la biblioteca estándar con gran cantidad de módulos y paquetes. Pero también existen módulos y paquetes de terceros. En esta sección vamos a formalizar esos conceptos, y a echar luz sobre cómo aprovechar estas funcionalidades.

Podemos pensar que tenemos tres anillos concéntricos de funcionalidad. Desde lo más cercano y fácilmente utilizable, hasta lo más lejano y con más preparación necesaria para ser aprovechada.

Empecemos por lo más cercano.

1.1.2.1. Integrada en Python

También denominada *builtin*, es la funcionalidad que está adentro de Python mismo como intérprete del lenguaje. Podemos separarla por un lado en aquella que tenemos a disposición por la sintaxis y semánticas mismas del lenguaje, y por el otro en la gran cantidad de objetos que podemos utilizar directamente.

La sintaxis y semántica del lenguaje en sí se encuentra detalladas en la Referencia del Lenguage [5], una sección de la documentación pensada para entender cómo funciona realmente el lenguaje y sus estructuras internas. Si estamos arrancando con Python no hace falta que nos involucremos con estos documentos ahora, pero es un paso obligado cuando el conocimiento del lenguaje se profundiza.

Python realmente no es un lenguaje, es la especificación de un lenguaje. La implementación de esa especificación más conocida es la que está escrita en C (a la que llamamos cPython cuando queremos marcar esa diferencia). Otras implementaciones son Jython (un Python escrito en Java), IronPython (en C#), MicroPython (que corre nativo en algunos microcontroladores), y notablemente PyPy, un Python escrito... ¡en Python!

Un ejemplo de funcionalidad parte del lenguaje mismo es la capacidad de poder llamar a una función con una cantidad variable de argumentos.

Por otro lado, los objetos que tenemos disponibles para utilizar directamente por estar integrados en el lenguaje están documentados al principio de la Referencia de la Biblioteca Estándar [6], agrupados en funciones, algunas constantes, tipos de datos y excepciones.

Como ejemplos de esto último podemos mencionar la función `len`, que nos permite saber el largo de una estructura, o el tipo de dato `int`, para representar enteros.

1.1.2.2. Parte de la Biblioteca Estándar

El resto de la Referencia de la Biblioteca Estándar [6] versa sobre aquella funcionalidad externa al intérprete de Python en sí mismo, pero disponible en cualquier instalación de Python.

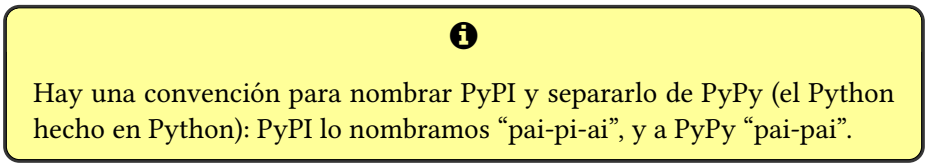
Está agrupada en distintos módulos, y para acceder a ella debemos importar esos módulos (más detalle sobre módulos y paquetes en 1.5.3).

Por ejemplo, para calcular el factorial de un número, podemos usar la función `factorial` del módulo `math`, que es el que debemos importar primero.

1.1.2.3. El resto del mundo

Si queremos funcionalidad de terceros, que no está integrada a Python o en la biblioteca estándar, siempre podemos instalar módulos disponibles en Internet.

La forma más sencilla y directa de hacer esto es utilizar el programa `pip`, que automáticamente descarga lo que indiquemos del Índice de Paquetes de Python (en inglés Python Package Index, al que normalmente conocemos como PyPI) y luego lo instala en nuestro sistema o algún entorno en particular.



Hay una convención para nombrar PyPI y separarlo de PyPy (el Python hecho en Python): PyPI lo nombramos “pai-pi-ai”, y a PyPy “pai-pai”.

Por supuesto, también podemos utilizar el administrador de paquetes de nuestro sistema (como `apt` en Debian/Ubuntu), utilizar pip para instalar paquetes provenientes de otros repositorios (como Github) o directamente podemos descargar los paquetes necesarios a mano e instalarlos.

A lo largo del libro iremos utilizando varios paquetes que se necesitan instalar de esta forma, los cuales estarán detallados al principio de cada capítulo. También vale la pena repasar el capítulo donde mostramos más en detalle las distintas alternativas para instalar paquetes **??**.

1.2. Editando, corriendo, e interpretando

En esta sección veremos cómo hacer nuestro primer programa en Python. Algo sencillo, pero que nos permitirá explorar distintas formas de editarlo y ejecutarlo, y otros conceptos aledaños.

1.2.1. Editando y ejecutando Python

Mencionamos arriba que podemos considerar que Python es interpretado, pero realmente es compilado. A fines prácticos, especialmente al arrancar, podemos considerar efectivamente que es interpretado: nosotros le pasamos el código fuente, y Python *lo ejecuta*.

¡Probemos eso! Agarremos cualquier editor (más adelante profundizamos ahí) y escribamos un programita muy simple, y lo grabamos en un archivo `hola.py`:

```
1 print('¡Hola mundo!')
```

Luego, en una terminal ejecutamos nuestro programa...

```
$ python3 hola.py
¡Hola mundo!
```

Es así de simple. Grabamos el archivo, lo ejecutamos, tenemos el resultado. Si queremos realizar modificaciones al archivo, volvemos a grabar, volvemos a ejecutar. Nada más. Este "ciclo corto" hace que iterar sobre la construcción de un programa sea muy eficiente en Python.

Por otro lado, ¿qué es eso de ir a la terminal para ejecutar un programa? Bueno, tenemos principalmente dos tipos de programas, con y sin interfaz gráfica. El que mostramos arriba no posee una interfaz gráfica, entonces lo usamos desde la terminal.

Si nuestro programa fuese de interfaz gráfica, podríamos ir y hacerle doble-click al programa y que se ejecute. En realidad también lo podemos hacer en un programa sin interfaz gráfica, pero todo lo que veríamos es una terminal que nuestro sistema abriría para ejecutar el programa, que se cerraría cuando este termine (y si el programa termina rápidamente, como el ejemplo de arriba, veríamos sólo un parpadeo).

Más adelante nos encargaremos de hacer programas con interfaz gráfica (ver Capítulo **??**), pero por ahora enfoquémosnos en saber al menos cómo ejecutar cualquier tipo de programa: desde la terminal.

Obviamente para ejecutar nuestro programa con Python necesitamos a Python instalado en nuestro sistema. Lo vamos a encontrar ya instalado en cualquier Linux o MacOS, pero vamos a necesitar instalarlo en Windows. En cualquier caso, todas las versiones para todos los sistemas están en la página oficial de descargas [7].

Como mostramos arriba, siempre podemos pasarle nuestro programa al intérprete de Python para que lo ejecute. Si queremos poder ejecutarlo directamente, quizás tengamos que realizar una acción extra, dependiendo de nuestro sistema. En Windows, el instalador mismo de Python asocia la extensión `.py` al intérprete de Python, entonces no necesitamos hacer nada.

En Linux y MacOS, por otro lado, tenemos que hacer que nuestro programa sea ejecutable e indicar en el mismo programa que debe ejecutarse con Python. Veamos esto en detalle. Primero modificamos el programa, agregando una linea muy particular al principio y luego una linea en blanco:

```
1 #!/usr/bin/python3
2
3 print('¡Hola mundo!')
```

Esa linea particular del principio tiene una estructura muy específica: arranca con estos dos caracteres `#!` (llamados *shebang*) que indican que a continuación está el programa que va a interpretar las líneas del archivo, `python3` en nuestro caso.

Luego hacemos ejecutable al programa, y lo corremos directamente:

```
$ chmod +x hola.py
$ ./hola.py
¡Hola mundo!
```

1.2.2. Editando Python

Python tiene una sintaxis tan sencilla y limpia que no se necesita demasiada ayuda del editor que estemos usando para escribir código.

Entonces, cualquier editor de texto que sea útil para escribir programas (NO Microsoft Word, por ejemplo) nos es suficiente. Ejemplos de este tipo son Vim, Emacs, Kate, Textmate o Notepad++.

Habiendo dicho eso, igualmente mucha gente prefiere desarrollar utilizando Entornos de Desarrollo Integrados (en inglés, *Integrated Development Environment*, lo que forma la sigla "IDE" que es la que se usa normalmente también en castellano). Esto es porque un IDE integra el editor (simple, como decíamos) con un navegador de archivos, herramientas de debugging, un interprete interactivo, y muchas funcionalidades más. Ejemplos de IDEs son PyCharm, VisualStudio, o Spyder.

Esas funcionalidades extras que proveen los IDEs son tan útiles que incluso se han ido montando herramientas sobre los editores simples para obtener algunas de ellas. Quizás el ejemplo más famoso de eso es la configuración de Fisa para Vim [8].

En cualquier caso, usar un editor simple más bien pelado o un IDE súper completo (o cualquier intermedio entre esos dos extremos) no deja de ser una elección personal. Nuestra recomendación es que las personas nuevas a la programación no se traben demasiado en arrancar con la herramienta ideal, que elijan una más o menos rápido y se pongan a programar, la elección del editor o IDE ideal para esa persona irá decantando con el tiempo.

Para elegir con qué arrancar siempre es bueno revisar la página de Python Argentina dedicada a editores e IDEs [9].

1.2.3. Usando módulos

Un concepto básico que necesitamos para arrancar es cómo utilizar los módulos que Python trae en la Biblioteca Estándar. Porque el programa que hicimos arriba usa la función integrada `print` pero enseguida vamos a tener que empezar a utilizar otras funciones no integradas.

Para usar un módulo primero tenemos que importarlo. La forma más directa de hacer esto es a través de la declaración `import`, y luego podremos acceder a los contenidos del módulo a través de la "notación punto" (este es uno de esos conceptos genéricos del lenguaje: siempre que queramos acceder a algo que está adentro de un objeto, podemos usar el punto: `.`.

Veamos un simple ejemplo.

```
1 import math
2
3 print('Raíz cuadrada de dos:', math.sqrt(2))
```

Ahí vemos que importamos el módulo `math` y luego llamamos a la función que nos calcula la raíz cuadrada usando el "punto": `math.sqrt(2)`.

Más detalles sobre módulos y distintas formas de importarlos y usarlos en 1.5.3.

1.2.4. El intérprete interactivo

En realidad no tenemos que escribir todo un programa para probar algo en Python. Uno de las mejores características del lenguaje es traer incorporado un intérprete interactivo.

Si ejecutamos `python3` sin pasarle ningún programa, Python automáticamente abrirá el intérprete interactivo, mostrándonos un *prompt* y esperando que escribamos algo:

```
$ python3
Python 3.8.2 (default, Jul 16 2020, 14:00:26)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Obviamente las primeras líneas pueden variar en función de la versión de Python que tengamos instalada, pero lo importante es ese >>> que nos habilita a empezar a escribir.

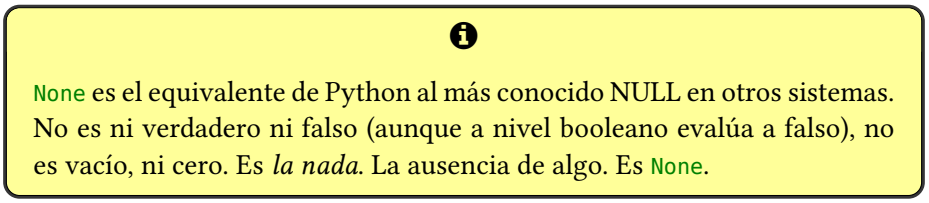
Llamamos "interactivo" a este modo del intérprete, porque Python compilará y ejecutará cada linea que escribamos, sin necesidad de otros pasos.

A lo largo de este libro veremos muchísimos ejemplos de código que son realizados directamente en el intérprete interactivo, y justamente nos podemos dar cuenta de eso por el prompt. Por ejemplo, reveamos como importar y usar un módulo, pero sin tener que escribir un programa para ello:

```
>>> import math
>>> print(math.sqrt(2))
1.4142135623730951
```

Vemos que la linea posterior al **import** es directamente el prompt donde escribimos el **print**: esto es porque Python no tiene nada que mostrarnos en el medio. Luego, como hacemos un **print**, la salida correspondiente la vemos directamente allí.

En realidad no es necesario hacer un **print** para ver un resultado en el intérprete interactivo, ya que este luego de ejecutar cada linea nos mostrará abajo el objeto resultado de esa linea, a menos que sea `None`.



`None` es el equivalente de Python al más conocido NULL en otros sistemas. No es ni verdadero ni falso (aunque a nivel booleano evalúa a falso), no es vacío, ni cero. Es *la nada*. La ausencia de algo. Es `None`.

Esto nos permite usar el intérprete de forma muy sencilla para explorar distintos comportamientos..

```
>>> 2 + 3
5
>>> len("hola")
4
```

Hay una diferencia significativa, sin embargo, entre la salida de **print** y lo que nos muestra el intérprete interactivo en cada resultado. Para construir la salida (convertir el objeto resultado a una representación textual a mostrar en la terminal) el **print** utiliza la función integrada `str`, que nos deja la representación más "simple y humana" del objeto, mientras que el intérprete interactivo usa la función integrada `repr`, que apunta a lograr la representación más "exacta" del objeto.

Vemos en el siguiente ejemplo como con el **print** corremos el riesgo de confundir el tipo de dato del resultado (¡parece un número!), mientras que con el `repr` es evidente que es una cadena de texto:

```
>>> print("23")
23
>>> "23"
'23'
```

En realidad el intérprete interactivo procesa la linea que acabamos de escribir cuando esa linea termina, lo que nos lleva a marcar la diferencia entre líneas lógicas y líneas reales: el intérprete interactivo se da cuenta cuando la linea todavía no terminó (aunque hayamos apretado ENTER) porque falta cerrarla lógicamente. Veamos un ejemplo de eso, donde la linea continúa porque falta cerrar el paréntesis del `print`, y veamos como el intérprete nos marca esa "línea continuación" con tres puntos abajo del prompt:

```
>>> print("Hola",
... 123)
Hola 123
```

1.2.5. Jupyter Notebook

El intérprete interactivo que trae Python por default no es el único que existe, sino que tenemos a disposición múltiples alternativas, especializadas para distintos fines.

Quizás el intérprete alternativo más popularizado es IPython [10], un shell interactivo que añade funcionalidades extras al intérprete interactivo incluido en Python, como resaltado de líneas y errores mediante colores, una sintaxis adicional para el shell, etc.

La funcionalidad núcleo de IPython fue evolucionando en el tiempo y se convirtió en una aplicación web llamada Jupyter Notebook. Por su baja barrera de entrada para interactuar con el mismo, soporte para integrar textos y gráficos entre las celdas de ejecución, y facilidad para la distribución del contenido, Jupyter Notebook se ha vuelto el intérprete interactivo de facto en el ámbito científico.

En las próximas secciones utilizaremos casi exclusivamente Jupyter Notebooks para los ejemplos mostrados en el libros (y en todos casos haremos referencia al notebook real, accesible en internet, para que puedan descargarlo, modificarlo y jugar con los ejemplos, que es una muy buena manera de aprender).

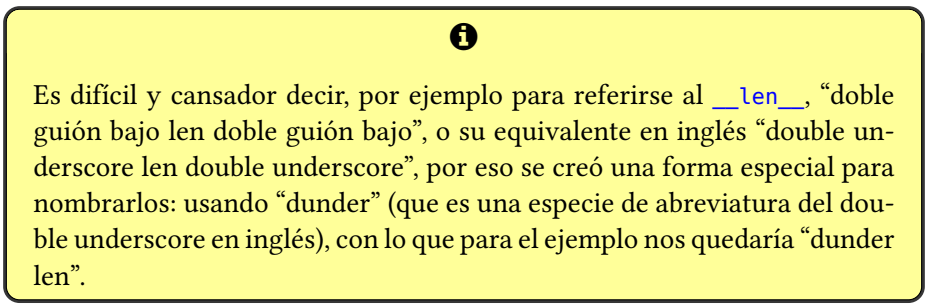
1.2.6. Explorando

Tanto el intérprete interactivo como Jupyter Notebook nos permite explorar el lenguaje y sus comportamientos. Esto es especialmente evidente si utilizamos las funciones integradas `dir` y `help`, que nos permiten ver los interiores de un objeto y directamente pedir ayuda en la terminal y ver su documentación.

En el siguiente ejemplo vemos como podemos descubrir qué atributos tiene el tipo de dato `list` y pedir ayuda sobre uno de ellos.

```
>>> dir([])
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__',
'__gt__', '__hash__', '__iadd__', '__imul__', '__init__', '__init_subclass__',
'__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__',
'__setattr__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__',
'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop',
'remove', 'reverse', 'sort']
>>> help([].count)

Help on built-in function count:

count(value, /) method of builtins.list instance
    Return number of occurrences of value.
```

Siempre que miremos dentro de los distintos objetos vamos a encontrar muchos atributos que empiezan y terminan con doble guión bajo. Estos son métodos especiales que permiten a los objetos integrarse y utilizar la sintaxis y semántica del lenguaje en sí (ya sean estos objetos integrados en Python, creados por nosotros o por terceros. Mientras estemos aprendiendo Python podemos ignorarlos tranquilamente, dejándolos para cuando avancemos con el lenguaje. Habiendo dicho eso, durante el libro mencionaremos y utilizaremos algunos, que explicaremos oportunamente.



Es difícil y cansador decir, por ejemplo para referirse al `__len__`, "doble guión bajo len doble guión bajo", o su equivalente en inglés "double underscore len double underscore", por eso se creó una forma especial para nombrarlos: usando "dunder" (que es una especie de abreviatura del double underscore en inglés), con lo que para el ejemplo nos quedaría "dunder len".

1.3. Tipos de datos

En esta sección mostraremos los tipos de datos más utilizados en Python.

No incluiremos *todos* los tipos de datos integrados en el lenguaje (ni mucho menos todos los presentes en la Biblioteca Estándar), sólo haremos foco en aquellos más utilizados y que les permitirá comenzar y realizar gran parte de todo lo que harán con Python.

1.3.1. Números

Comenzamos con el tipo de datos más básico en todos los lenguajes: los números enteros, cuyo tipo de datos es `int`.

Las operaciones básicas son las mismas que en cualquier otro lado y no tenemos ninguna sorpresa, con los únicos detalles a destacar que la división de números enteros da un `float` (el punto flotante binario, que veremos a continuación), y que el operador para la exponenciación es `**`.

Cell 01

```
3 + 5
```

```
8
```

Cell 02

```
2 - 8
```

```
-6
```

Cell 03

```
4 * -8
```

```
-32
```

CELL 04

```
32 / 5
```

```
6.4
```

CELL 05

```
5 ** 2
```

```
25
```

Complementando la división, Python incluye un operador para la división con el resultado truncado, y el módulo, e incluso una función integrada que da ambos resultados al mismo tiempo:

CELL 06

```
32 // 5
```

```
6
```

CELL 07

```
32 % 5
```

```
2
```

CELL 08

```
divmod(32, 5)
```

```
(6, 2)
```

Más allá de estas operaciones particulares, el rasgo más relevante a destacar en los enteros de Python es que no tienen límite (porque no están atados a ninguna representación en particular), entonces no hace falta que nos preocupemos por ese detalle:

CELL 09

```
2529 ** 15 * 1834
```

```
203063455156707669488854164141449297202835448823297546 6
```

Los números enteros en Python también funcionan como secuencias de bits, y más allá que usemos la representación decimal para expresarlos, o en binario, o en hexadecimal, simplemente son números enteros. Escribirlos en otras bases es sencillo, y para verlos en esas representaciones tenemos funciones integradas:

CELL 10

```
0b1001010
```

```
74
```

CELL 11

```
0x3f
```

```
63
```

Cell 12

```
hex(63)
```

```
'0x3f'
```

Cell 13

```
bin(74)
```

```
'0b1001010'
```

Python tiene operadores para trabajar con los enteros como secuencias de bits: el "and" (&), el "or" (|), el "xor" (^), y los "shift" a izquierda y derecha (<< y >>).

Cell 14

```
bin(0b1001 & 0b0001)
```

```
'0b1'
```

Cell 15

```
bin(0b1001 | 0b0001)
```

```
'0b1001'
```

Cell 16

```
bin(0b1001 ^ 0b0001)
```

```
'0b1000'
```

Cell 17

```
bin(0b1100 << 2)
```

```
'0b110000'
```

Cell 18

```
bin(0b1100 >> 2)
```

```
'0b11'
```

El punto flotante binario (`float`) de Python, por otro lado, es exactamente el punto flotante ejecutado en el procesador, y tiene el mismo comportamiento que en los otros lenguajes:

Cell 19

```
2.35 * 28
```

```
65.8
```

Cell 20

```
15 + 2.3
```

```
17.3
```

CELL 21

```
95 / 17
```

```
5.588235294117647
```

CELL 22

```
2.3 ** 15
```

```
266635.23546439095
```

Así y todo siendo uno de los tipos de datos más utilizados en todos los lenguajes, hay ciertas trampas del punto flotante binario que es imperioso conocer, y a los que les dedicamos un capítulo separado en ??.

Uno de esos detalles es que no debemos utilizar `float` para manejar dinero. De esa limitación nació la implementación del punto flotante decimal, que podemos encontrar en el módulo `decimal` de la Biblioteca Estándar.

CELL 23

```
from decimal import Decimal

Decimal(23) * 53
```

```
Decimal('1219')
```

CELL 24

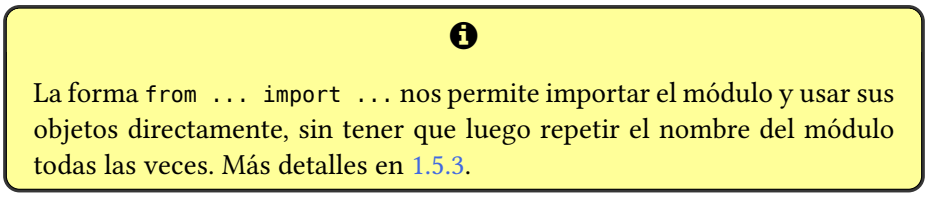
```
Decimal(15.4) + 87
```

```
Decimal('102.40000000000000003552713679')
```

CELL 25

```
Decimal('15.4') + 87
```

```
Decimal('102.4')
```



La forma `from ... import ...` nos permite importar el módulo y usar sus objetos directamente, sin tener que luego repetir el nombre del módulo todas las veces. Más detalles en 1.5.3.

Como vemos en el ejemplo, no es lo mismo construir el Decimal a partir de la representación textual de un número "con coma" que de un `float`; volvemos a hacer referencia al capítulo ?? donde nos adentramos en estos detalles.

Relacionado, tenemos la posibilidad de directamente utilizar fracciones si así fuese necesario, a través del módulo `fractions`:

CELL 26

```
from fractions import Fraction

Fraction(1, 3) * 6
```

```
Fraction(2, 1)
```

CELL 27

```
Fraction(27, 5) ** 2
```

```
Fraction(729, 25)
```

CELL 28

```
Fraction(2, 3) + Fraction(6, 9)
```

```
Fraction(4, 3)
```

Finalmente, debemos mencionar el tipo de datos `complex`, integrado en el lenguaje, para manejar números complejos. En verdad tenemos soporte en la sintaxis para números imaginarios (agregándoles una `j` al final), y con ello armamos los complejos directamente:

CELL 29

```
3j
```

```
3j
```

CELL 30

```
3j + 2
```

```
(2+3j)
```

CELL 31

```
(3j + 2) * (7 + 2.5j)
```

```
(6.5+26j)
```

CELL 32

```
1j ** 2
```

```
(-1+0j)
```

CELL 33

```
import math

x = 1.5
math.cos(x) + 1j * math.sin(x) == math.e ** (x * 1j)
```

```
True
```

Las clases de cada tipo de dato funcionan como constructores y al mismo tiempo como conversores entre ellos mismos:

CELL 34

```
int()
```

```
0
```

Cell 35

```
int(9.2)
```

```
9
```

Cell 36

```
float(-7)
```

```
-7.0
```

1.3.2. Cadenas

Las cadenas constituyen otro de los tipos de datos básicos que están presentes en infinidad de lenguajes. En Python no tienen nada de especial a primera vista, pero vamos a ir explorando algunas particularidades más adelante.

Por lo pronto hagamos la definición formal: las cadenas en Python son secuencias de caracteres Unicode delimitadas por comillas dobles o simples indistintamente.

En el siguiente ejemplo vemos casos de uno y otro delimitador, especialmente para el caso donde necesitamos usar uno de ellos como carácter dentro de la cadena:

Cell 01

```
"Hola mundo"
```

```
'Hola mundo'
```

Cell 02

```
'Python :)'
```

```
'Python :)'
```

Cell 03

```
"Let's rock!"
```

```
"Let's rock!"
```

También como en tantos otros lenguajes, la barra invertida tiene el propósito de escapar ciertos caracteres especiales (y es un último recurso para usar los delimitadores dentro de la cadena). Hay que tener en cuenta siempre que la barra invertida sirve para escaparse a sí misma, y que los caracteres especiales suelen mostrarse distinto en el modo `str` (representación más humana) que en el modo `repr` (representación más exacta):

Cell 04

```
"Hola\nmundo"
```

```
'Hola\nmundo'
```

Cell 05

```
print("Hola\nmundo")
```

```
Hola
mundo
```

Cell 06

```
"foo \" bar"
```

```
'foo " bar'
```

Cell 07

```
"foo \" bar ' baz"
```

```
'foo " bar \' baz'
```

Cell 08

```
'foo \\" bar'
```

```
'foo \\" bar'
```

Cell 09

```
print('foo \\" bar')
```

```
foo \" bar
```

Python tiene otro delimitador para cadenas: la triple comilla (ya sea doble o simple), que nos permite escribir una cadena a través de múltiples líneas:

Cell 10

```
mensaje = """
Hola
    mundo
        :)
"""
print(mensaje)
```

```

Hola
    mundo
        :)

```

Cell 11

```
mensaje
```

```
'\nHola\n    mundo\n        :)\n'
```

Las operaciones básicas a realizar sobre las cadenas están integradas en el lenguaje: saber el largo, concatenar y repetir:

Cell 12

```
len("Hola")
```

```
4
```

Cell 13

```
"Ho" + 'la'
```

```
'Hola'
```

Cell 14

```
'Na' * 5 + " Batman!"
```

```
'NaNaNaNaNa Batman!'
```

Las cadenas tienen también muchos métodos para trabajar con las mismas, veamos algunos de ellos (¡sólo ejemplos! les recomendamos revisar todos los métodos en [11], ya que son muy útiles en el día a día):

Cell 15

```
"Moño".upper()
```

```
'MOÑO'
```

Cell 16

```
"python is a great language".title()
```

```
'Python Is A Great Language'
```

Cell 17

```
"python is a great language".index("great")
```

```
12
```

Cell 18

```
"12345".isdigit()
```

```
True
```

Cell 19

```
"\t\t corrido  \n".strip()
```

```
'corrido'
```

Un método en particular de gran utilidad es el `format`, que permite armar cadenas reemplazando valores, lo cual es mucho más legible y ofrece más control que andar concatenando cadenas individuales. Más allá de estos ejemplos puntuales que mencionamos a continuación, el sistema de formateo en cuestión es muy poderoso, y vale la pena al menos sobrevolar la documentación [12].

Cell 20

```
"Tiempo estimado: {:.2f}s".format(1.34844)
```

```
'Tiempo estimado: 1.35s'
```

Cell 21

```
"Valor para muestra {:05d}: |{:>13s}|".format(7, "failure")
```

```
'Valor para muestra 00007: |      failure|'
```

Notablemente, no mencionamos ningún método para acceder a un carácter de la cadena, o a una subcadena de la misma. Esto se debe a que dicha funcionalidad está integrada en el lenguaje y es la misma para todas las secuencias.

La forma de acceder a un carácter de la secuencia es con los corchetes, escribiendo directamente la posición:

CELL 22

```
cad = "Hola mundo"
cad[0]
```

```
'H'
```

CELL 23

```
cad[2]
```

```
'l'
```

CELL 24

```
cad[-2]
```

```
'd'
```

En el ejemplo vemos que la primer posición de la secuencia es la número 0, y que si utilizamos números negativos se empieza a contar desde el final (lo cual es muy práctico, porque no hay que calcular la posición que queremos restándole algo al largo de la cadena).

Para subcadenas es también con corchetes, pero usamos dos valores, desde y hasta (con valores por defecto "desde el principio" o "hasta el final" si no se incorporan), y también soporta valores negativos, como indicábamos arriba:

CELL 25

```
cad = "Hola mundo"
cad[2:6]
```

```
'la m'
```

CELL 26

```
cad[:6]
```

```
'Hola m'
```

CELL 27

```
cad[2:]
```

```
'la mundo'
```

CELL 28

```
cad[-3:]
```

```
'ndo'
```

Esta operación de tomar la subcadena de una cadena en inglés se llama *slicing*, y en general usamos ese término también en castellano (porque *rebanar* cadenas nunca terminó prendiendo).

Entender qué carácter se accede por posición es sencillo (sólo tenemos que acordarnos contar desde 0), pero en el caso de los slices es más difícil, hasta que nos acostumbramos. Una buena regla mnemotécnica es pensar que también arranca desde 0, pero lo que se cuentan son las "separaciones entre las letras". Más allá de cómo nos acordemos, la regla que tiene Python para numerar en los slices tiene dos propiedades muy útiles: por un lado si hacemos `cad[x:y]` con

valores positivos el largo de la subcadena va a ser `y - x`, y por el otro es sencillo separar una cadena en dos, porque se usa el mismo número, o sea que `cad[:x] + cad[x:]` nos termina dando la misma `cad`.

También como todas las secuencias, si queremos tomar elementos de la misma pero con un determinado paso, podemos utilizar un tercer valor entre los corchetes:

CELL 29

```
cad = "Hola mundo, chau mundo\n"
cad[:-1:2]
```

```
'Hl ud,ca ud'
```

En el ejemplo usamos varias propiedades al mismo tiempo. Por un lado el *desde* lo dejamos en blanco, para que tome el inicio por default, luego el *hasta* lo usamos negativo porque nos interesaba obviar el *newline* del extremo derecho, y finalmente el tercer valor, el paso, en dos, para ir agarrando cada dos caracteres.

Como con los números, podemos usar el constructor `str` de la cadena para convertir desde otros tipos de datos (así como podemos también usar los constructores de números para pasar de cadenas a enteros o flotantes):

CELL 30

```
str(2.3)
```

```
'2.3'
```

CELL 31

```
float("1.44")
```

```
1.44
```

CELL 32

```
int("23")
```

```
23
```

CELL 33

```
int("3F", base=16)
```

```
63
```

Cuando mencionamos que los delimitadores de las cadenas eran la comilla doble, o simple, o sus variantes "triple", evitamos mencionar que en todos estos casos se puede poner un prefijo para cambiar el tipo de cadena definida. Por default si no especificamos nada (o si usamos la letra *u*, por compatibilidad con versiones viejas de Python), la cadena será de tipo Unicode, lo que implica que será una secuencia de caracteres Unicode, con lo cual podemos escribir caracteres con acento, o en otros idiomas:

CELL 34

```
"El camión llegará mañana"
```

```
'El camión llegará mañana'
```

Cell 35

```
"El valor de π es aproximadamente 3.14"
```

```
'El valor de π es aproximadamente 3.14'
```

Cell 36

```
"¡Hola! привет"
```

```
'¡Hola! привет'
```

Tengamos en cuenta que considerar a los caracteres como Unicode, más allá de su posible representación como bytes, nos permite trabajar con herramientas de transformación de texto sin mayor inconveniente:

Cell 37

```
m = "moño"
len(m)
```

```
4
```

Cell 38

```
m.upper()
```

```
'MOÑO'
```

Si necesitamos trabajar con bytes directamente, podemos tener cadenas de bytes (donde no es más una secuencia de caracteres, sino una secuencia de bytes) si prefijamos la cadena con la letra *b*. En estos casos es muy útil la notación `\x` para ingresar el valor del byte en hexadecimal, como mostramos en el siguiente ejemplo:

Cell 39

```
val = b"ab\x00\xffcd"
val
```

```
b'ab\x00\xffcd'
```

Cell 40

```
len(val)
```

```
6
```

Cell 41

```
val[3]
```

```
255
```

Es importante entender esta distinción no sólo porque las cadenas de bytes eran el default en versiones viejas de Python (y nos podemos cruzar con algún código así) sino también porque eventualmente necesitaremos convertir las cadenas Unicode a cadenas de bytes, ya que es la única manera de mandar cadenas a través de la red o grabarlas en disco.

Estas conversiones las hacemos con los métodos `encode` y `decode`. Aunque en el siguiente ejemplo parece simple, el tema de convertir de un lado para el otro le trae muchos dolores de cabeza a la mayoría de los programadores (más allá del lenguaje en que programen); les reco-

mendamos que si quieren adentrarse en el tema vean esta charla por uno de los autores del libro [13].

CELL 42

```
mu = "moño"
mu
```

```
'moño'
```

CELL 43

```
mb = mu.encode("utf8")
mb
```

```
b'mo\xc3\xb1o'
```

CELL 44

```
mb.decode("utf8")
```

```
'moño'
```

Otro tipo de cadena usada frecuentemente es la de tipo *raw* (término que usamos en inglés, porque decir que son "sin procesar" o "crudas" es raro), donde la diferencia con las cadenas comunes es que la barra invertida NO funciona como carácter de escape (lo cual es especialmente útil al escribir expresiones regulares, tema que por respeto a les lectores NO tocaremos en el libro), sino que es tratada como un carácter normal:

CELL 45

```
r"\n"
```

```
'\\n'
```

CELL 46

```
len(r"\n")
```

```
2
```

Finalmente, en las versiones más modernas de Python tenemos un tipo de cadena que se formatea automáticamente con las variables del entorno, sin tener que llamar explícitamente a `format`, permitiendo incluso expresiones y algunos detalles más:

CELL 47

```
t = 1.34844
f"Tiempo estimado: {t:.2f}s"
```

```
'Tiempo estimado: 1.35s'
```

CELL 48

```
vals = [1, 4, 2, 1, 7]
f"Proceso completo! init={vals[0]} promedio={sum(vals)/len(vals)}"
```

```
'Proceso completo! init=1 promedio=3.0'
```

1.3.3. Listas

Las listas son secuencias de objetos, se delimitan con corchetes, y cada objeto (sus elementos internos, que pueden ser cualquier cosa) se separan con comas.

Se acceden como cualquier secuencia, exactamente como vimos antes con las cadenas 1.3.2, e incluso tienen la misma forma de repetirlas o concatenarlas:

CELL 01

```
lista = [1, "a", 5.6, "foo"]
lista
```

```
[1, 'a', 5.6, 'foo']
```

CELL 02

```
len(lista)
```

```
4
```

CELL 03

```
lista[1]
```

```
'a'
```

CELL 04

```
lista[2:]
```

```
[5.6, 'foo']
```

CELL 05

```
lista * 2
```

```
[1, 'a', 5.6, 'foo', 1, 'a', 5.6, 'foo']
```

CELL 06

```
lista + [1, 2]
```

```
[1, 'a', 5.6, 'foo', 1, 2]
```

La gran diferencia con las cadenas a nivel comportamiento (más allá que unas son secuencias de caracteres y las otras son secuencias de cualquier objeto Python) es que mientras las cadenas son "inmutables", las listas (como muchos otros tipos de datos) son "mutables", o sea que pueden cambiar.

El detalle fino de esto se explica un poco más adelante 1.3.5, pero veamos como para las listas efectivamente tenemos métodos que permiten modificarlas:

CELL 07

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

```
[1, 2, 3]
```

Cell 08

```
lista.append(4)
lista
```

```
[1, 2, 3, 4]
```

Cell 09

```
otra = [5, 6]
lista.extend(otra)
lista
```

```
[1, 2, 3, 4, 5, 6]
```

¡Y no sólo agregarle elementos! También podemos reemplazar algún elemento o toda una parte:

Cell 10

```
lista
```

```
[1, 2, 3, 4, 5, 6]
```

Cell 11

```
lista[1] = 99
lista
```

```
[1, 99, 3, 4, 5, 6]
```

Cell 12

```
lista[2:] = [0, 0]
lista
```

```
[1, 99, 0, 0]
```

También podemos borrar elementos, no sólo con el método que nos permite especificar cual elemento en sí queremos borrar, sino también con la sentencia `del`.

Cell 13

```
lista
```

```
[1, 99, 0, 0]
```

Cell 14

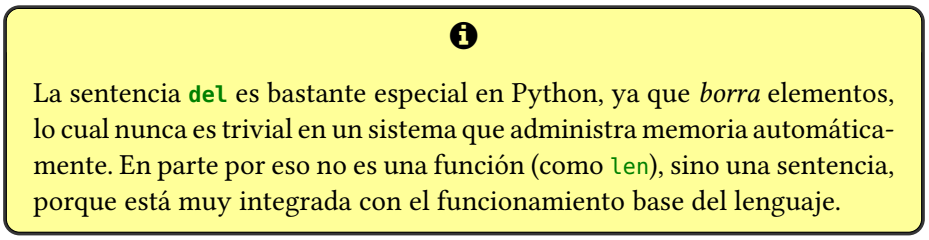
```
lista.remove(99)
lista
```

```
[1, 0, 0]
```

Cell 15

```
del lista[1]
lista
```

```
[1, 0]
```



La sentencia **del** es bastante especial en Python, ya que *borra* elementos, lo cual nunca es trivial en un sistema que administra memoria automáticamente. En parte por eso no es una función (como `len`), sino una sentencia, porque está muy integrada con el funcionamiento base del lenguaje.

Como en los otros casos, el constructor de las listas nos es bastante útil para crear listas a partir de otros objetos. `list` construye una lista a partir de los elementos del iterable que reciba:

Cell 13

```
lista
```

```
[1, 99, 0, 0]
```

Cell 14

```
lista.remove(99)
lista
```

```
[1, 0, 0]
```

Cell 15

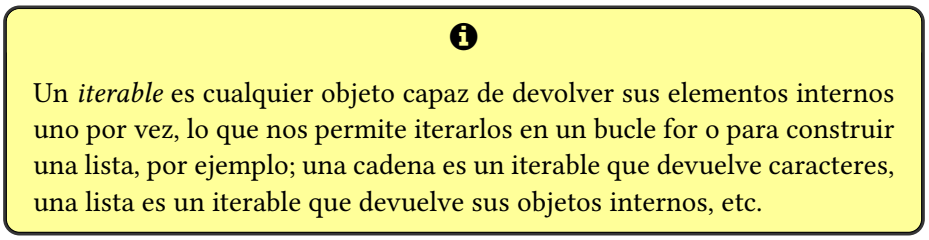
```
del lista[1]
lista
```

```
[1, 0]
```

Cell 16

```
list("abcd")
```

```
['a', 'b', 'c', 'd']
```



Un *iterable* es cualquier objeto capaz de devolver sus elementos internos uno por vez, lo que nos permite iterarlos en un bucle for o para construir una lista, por ejemplo; una cadena es un iterable que devuelve caracteres, una lista es un iterable que devuelve sus objetos internos, etc.

1.3.4. Tuplas

Las tuplas también son secuencias de objetos, y se definen separando estos elementos con coma (como en la definición de las listas), y opcionalmente rodeándolas con paréntesis, por claridad.

Cell 01

```
(1, 'a', 2.3)
```

```
(1, 'a', 2.3)
```

CELL 02

```
1, 5, 'xx'
```

```
(1, 5, 'xx')
```

CELL 03

```
t = (5,)
t
```

```
(5,)
```

CELL 04

```
len(t)
```

```
1
```

CELL 05

```
t[0]
```

```
5
```

Una diferencia fundamental con las listas es que las tuplas son inmutables, y en general una buena recomendación para decidir su uso en un caso u otro es entender si para los objetos internos es importante su cantidad y posición. Por ejemplo, la lista es mejor para guardar los archivos que hay en un directorio (más allá del orden en sí, el archivo no es más o menos archivo si está antes o después, y en el directorio podemos tener 0, 3, 27, 1000 archivos), en cambio la tupla es mejor para guardar coordenadas (si es en el plano vamos a tener siempre una tupla de dos elementos, ni uno ni tres, y es importante si un número está primero o segundo, ya que son ejes distintos).

En realidad la diferencia es aún más profunda: las tuplas forman parte del núcleo más central del lenguaje, ya que se usa en infinidad de detalles internos. Por ejemplo, si recordamos la función interna `divmod` que explicamos cuando vimos números, veremos que esa función devuelve dos números... en verdad devuelve una tupla, que por otra característica de Python ("desempaquetado de tuplas", que también tendemos a denominar en inglés como "tuple unpacking") podemos usar directamente como si fueran dos objetos:

CELL 06

```
divmod(60, 7)
```

```
(8, 4)
```

CELL 07

```
a, b = divmod(60, 7)
print("a={}, b={}".format(a, b))
```

```
a=8, b=4
```

Obviamente necesitamos la misma cantidad de elementos en la izquierda y la derecha del igual, para que el "desempaquetado" funcione, aunque tenemos la posibilidad de usar un "expansor" para que tome múltiples argumentos:

CELL 08

```
a, b, c = 1, 2
```

```
ValueError                                Traceback (most recent call last)
<ipython-input-8-9dbc59cfd6c6> in <module>
----> 1 a, b, c = 1, 2

ValueError: not enough values to unpack (expected 3, got 2)
```

CELL 09

```
a, b, c = 1, 2, 3, 4, 5, 6
```

```
ValueError                                Traceback (most recent call last)
<ipython-input-9-13682f951420> in <module>
----> 1 a, b, c = 1, 2, 3, 4, 5, 6

ValueError: too many values to unpack (expected 3)
```

CELL 10

```
a, b, *c = 1, 2, 3, 4, 5, 6
print("a={}, b={}, c={}".format(a, b, c))
```

```
a=1, b=2, c=[3, 4, 5, 6]
```

CELL 11

```
a, *b, c = 1, 2, 3, 4, 5, 6
print("a={}, b={}, c={}".format(a, b, c))
```

```
a=1, b=[2, 3, 4, 5], c=6
```

Una vez más, el constructor es la herramienta principal para convertir entre tipos:

CELL 12

```
tuple("foobar")
```

```
('f', 'o', 'o', 'b', 'a', 'r')
```

CELL 13

```
tuple([2, None, 0.1])
```

```
(2, None, 0.1)
```

1.3.5. Pensando como un pythonista

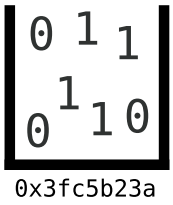
Hay una diferencia fundamental en cómo Python maneja los objetos internamente en la ejecución de un programa con respecto a otros lenguajes, y es imperativo entenderla porque nos explica mucho del funcionamiento del lenguaje.

Python no tiene *variables*, y en consecuencia esas variables no tienen *valores*. Estos son términos y conceptos que provienen de otros lenguajes de más bajo nivel (es decir, más cercanos al procesador), y no se aplican en Python.

Python tiene *objetos*, y usamos *nombres* para hacer referencia a esos objetos. Es verdad que a veces por facilismo o sobresimplificación usamos las palabras "variable" y "valor" también cuando hablamos de un programa en Python o el funcionamiento del lenguaje, pero eso no nos tiene que

confundir sobre el funcionamiento real.

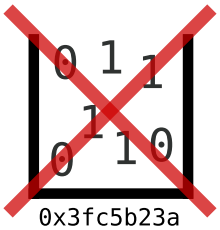
Veamos la diferencia. Otros lenguajes, como C por ejemplo, sí poseen el concepto de *variable*, como lugar donde se guarda un valor. Ese lugar es en memoria, y el valor que se guarda son los bits en esa posición de memoria.



En estos lenguajes es necesario declarar cómo se interpretan esos bits en esa posición de memoria. Por ejemplo, varios bytes seguidos pueden ser varias letras, o un número entero, e incluso podemos decirle al lenguaje que nos muestre esos bits de una manera u otra:

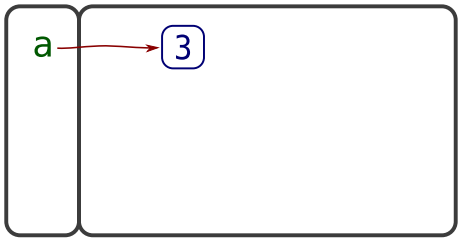
```
int a = 7303014;
printf("%i %s", a, (char *)&a);

  -->  7303014 foo
```

En cambio, en Python no seguimos esa filosofía.

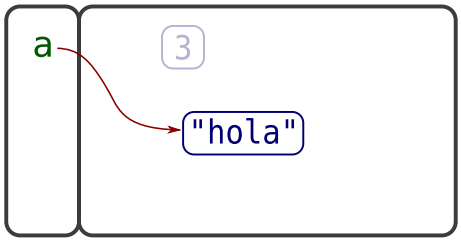


En Python manejamos directamente objetos. Estos objetos son de un tipo específico, y no pueden cambiar de tipo (“tipado fuerte”). Para referenciar esos objetos en memoria usamos nombres.

Estos nombres pueden apuntar a uno u otro objeto, y no están asociados al tipo del objeto que vinculan. Entonces, no hace falta decir que el nombre `a` va a ser un entero o una cadena, por ejemplo, porque el tipo pertenece al objeto que es nombrado. Cuando decimos que `a = 3` sólo estamos creando un objeto en memoria de tipo entero (el 3) y a ese objeto lo estamos referenciando usando el nombre `a`.



Si luego escribimos que `a = "hola"`, estamos creando otro objeto en memoria, este de tipo cadena, con el valor `"hola"`, y estamos usando el mismo nombre `a` para referenciarlo. No es que `a` haya cambiado de tipo, sólo apunta a otro objeto.



Obviamente, como `a` apunta al segundo objeto, ya no apunta al primero. Si ese objeto no es accesible desde ningún otro lado, para todo propósito nosotros podemos considerar que ya no existe (eventualmente Python lo eliminará y liberará memoria, pero esto es algo que podemos ignorar tranquilamente ya que Python administra la memoria por nosotros).

En los diagramas mostrados arriba vemos dos áreas diferentes. El grande es el espacio de objetos (a grandes rasgos lo que llamamos "la memoria", sin entrar en detalle cual sección de memoria en particular o cómo es manejada por Python). La columna de la izquierda, donde por ahora tenemos el nombre `a`, es llamada "espacio de nombres", una sección en particular (de la memoria, obviamente, porque de última todo está en memoria, pero que no se nos mezcle con la otra parte genérica) donde guardamos los nombres, que son simplemente cadenas. En el espacio de nombres no podemos almacenar otra cosa que nombres, y estos nombres siempre apuntan a objetos "en la memoria" (no pueden apuntar a otros nombres en el espacio de nombres).

Python tiene un espacio de nombres "global", que está siempre disponible durante la ejecución del programa y accesible de cualquier lado, y va creando otros espacios de nombres en diferentes momentos y con una accesibilidad limitada. Por ejemplo, cuando se ejecuta una función esta tiene su propio "espacio de nombres local", diferente al global y diferente a los espacios de nombres de otras funciones, que es lo que permite que cada función pueda usar un mismo nombre apuntando a diferentes objetos y no entren en conflicto.

En los diagramas, además de las dos secciones, y el nombre en una y el objeto en otra, vemos una flecha que va del nombre al objeto. Esta flecha representa justamente que ese nombre referencia a dicho objeto. Por eso cuando en Python escribimos `a = "hola"` estamos realmente *vinculando* el nombre `a` con el "objeto cadena con valor `"hola"`" (en inglés el verbo es "bind").

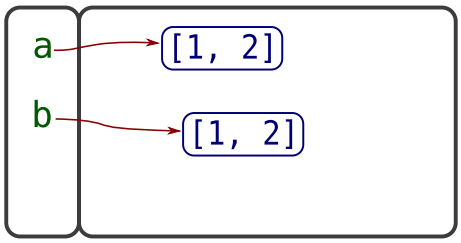
Retomemos el uso de estos diagramas mostrando los nombres y su vinculación a los objetos, y la diferencia fundamental entre objetos mutables e inmutables, que en conjunto son claves para entender cómo funciona Python internamente (para que eventualmente logremos "pensar como un Pythonista").

Vayamos entonces con un ejemplo más complejo.

Ahora en los siguientes dos pasos vinculamos primero el nombre `a` a una lista con dos números, y luego `b` a otra lista con los mismos dos números.

CELL 01

```
a = [1, 2]
b = [1, 2]
```



(El diagrama tiene una simplificación, en pos de la legibilidad: en verdad la lista no tiene a los números "adentro", sino que los números son otros objetos en la memoria, y desde cada posición de la lista se los referencia; vamos a ver esto mismo en los próximos diagramas cuando la complejidad lo amerite, pero en este caso solamente para los números no vale la pena.)

Vemos que en memoria tenemos dos objetos lista con el mismo contenido. Podemos adivinar que si le preguntamos a Python si ambos nombres apuntan a objetos iguales, nos dirá que sí, pero si le preguntamos si los dos nombres apuntan *al mismo objeto*, nos dirá que no.

CELL 02

```
a == b
```

```
True
```

CELL 03

```
a is b
```

```
False
```

El primer comparador es el de "igualdad", mientras que el segundo es de "identidad". Vamos a ver más comparadores luego cuando veamos el **`if`** en 1.4.1.

Modifiquemos ahora una de las listas.

CELL 04

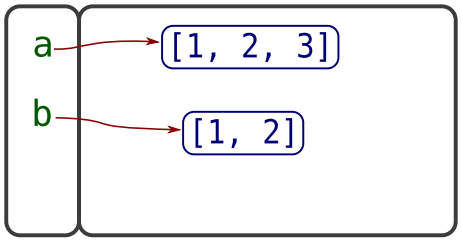
```
a.append(3)
a
```

```
[1, 2, 3]
```

CELL 05

```
b
```

```
[1, 2]
```



Vemos que la lista `a` ahora tiene un elemento más, mientras que la `b` sigue con su estado anterior, como podíamos esperar.

Intentemos algo nuevo, ahora usemos el = entre dos nombres. Acá es importante que recordemos que no podemos vincular un nombre a otro nombre (apuntar con la flechita de un nombre a otro nombre), sino que lo que termina sucediendo es que ese nombre nuevo queda vinculado al objeto que apuntaba el otro nombre.

Cell 06

```
c = b
c
```

```
[1, 2]
```

Cell 07

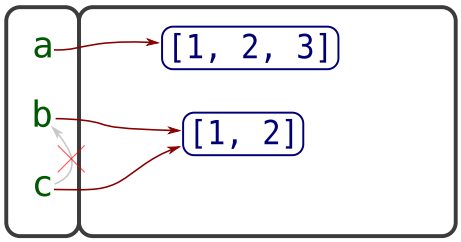
```
c == b
```

```
True
```

Cell 08

```
c is b
```

```
True
```



Es por eso que si ahora comparamos la igualdad y la identidad entre `c` y `b` Python contestará verdadero en ambos casos, porque ambos nombres apuntan al *mismo* objeto.

Si ahora modificamos la lista `c`, vemos que también se modifica `b`. Habiendo escrito eso, hagamos el ejercicio de entender que esa frase es inexacta, y casi tramposa. Porque `c` o `b` no son listas, son nombres que apuntan a una lista, la misma, entonces tampoco tiene sentido decir que "también se modifica". Entonces, escrbiendo esa frase correctamente, podríamos decir que "podemos modificar la lista y ver ese cambio, usando tanto un nombre como el otro", que es lo que vemos en el siguiente paso.

Cell 09

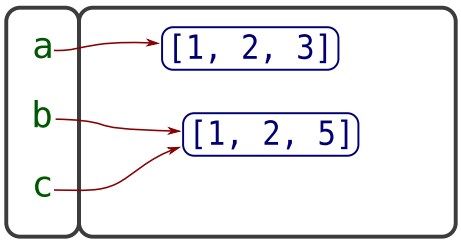
```
c.append(5)
c
```

```
[1, 2, 5]
```

Cell 10

```
b
```

```
[1, 2, 5]
```



Arranquemos otro ejemplo.

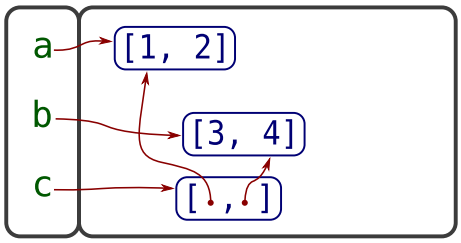
Volvemos a tener dos listas con números, y ahora también armamos una tercer lista que incluye las primeras dos. En verdad las dos primeras listas no están "adentro" de la primera, sino que son referenciadas en cada posición. Esto es exactamente lo mismo que mencionábamos arriba acerca de que dibujábamos a los números adentro de la lista pero en verdad eran objetos también en la zona de memoria, referenciados desde la lista.

En este caso en el dibujo seguimos con la simplificación de dibujar a los números adentro de la lista, pero somos más precisos con la tercer lista apuntando a las primeras dos.

Cell 11

```
a = [1, 2]
b = [3, 4]
c = [a, b]
c
```

```
[[1, 2], [3, 4]]
```



Si ahora modificamos la lista que llamamos `a`, obviamente vamos a ver que esa lista está cambiada, pero también vamos a ver reflejado ese cambio si miramos `c`, ya que estamos hablando en definitiva de la misma lista vista directamente o a través de una posición de la otra lista.

Cell 12

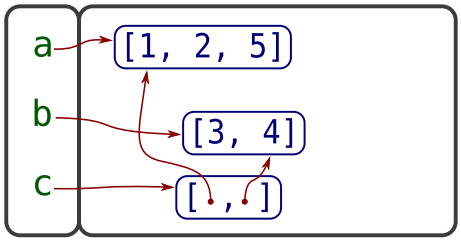
```
a.append(5)
a
```

```
[1, 2, 5]
```

Cell 13

```
c
```

```
[[1, 2, 5], [3, 4]]
```



Por otro lado, si hacemos `b = "foo"` tenemos que entender que no estamos modificando el objeto referenciado por `b` sino que realmente estamos revinculando el nombre `b` a otro objeto, y por lo tanto el objeto anterior (el que dejamos de referenciar cuando apuntamos `b` a otro lado) queda intacto.

Cell 14

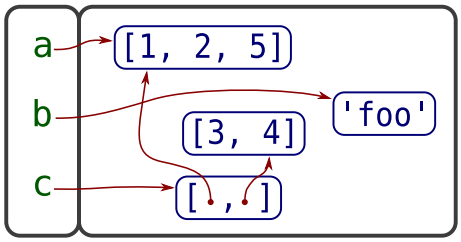
```
b = "foo"
b
```

```
'foo'
```

Cell 15

```
c
```

```
[[1, 2, 5], [3, 4]]
```



A diferencia del ejemplo que teníamos al principio de la subsección, donde el objeto "viejo" quedaba sin uso y decíamos que Python eventualmente lo iba a eliminar para liberar memoria, en este caso el objeto que antes era `b` no queda totalmente desreferenciado, sigue siendo apuntado por `c`, con lo cual permanecerá en memoria sin problemas.

E incluso podríamos volver a asignarle otro nombre, y seguir interactuando con el objeto directamente a través del nuevo nombre:

CELL 16

```
d = c[1]
d
```

```
[3, 4]
```

CELL 17

```
d.append(7)
d
```

```
[3, 4, 7]
```

CELL 18

```
c
```

```
[[1, 2, 5], [3, 4, 7]]
```

Como nota final, cabe destacar que el comparador de identidad `is`, con la excepción de cuando preguntamos `is None` o `is not None` (que queda bien en inglés), se usa en contadísimas ocasiones, en general necesitamos comparar por igualdad con el `==`.

1.3.6. Conjuntos

Los conjuntos son contenedores de objetos, se delimitan con llaves, y cada objeto (sus elementos internos) se separan con comas.

A diferencia de otros contenedores que vimos previamente, los conjuntos no son secuencias, ya que los elementos dentro del conjunto no tienen un orden en particular. Es hasta inexacto decir que "están desordenados", porque directamente este tipo de dato no tiene el concepto de orden.

Las propiedades más interesantes de los conjuntos son las matemáticas: cada elemento puede estar solamente una vez, y además podemos realizar intersecciones, uniones y diferencias entre ellos.

CELL 01

```
frutas = {'manzana', 'naranja', 'sandía', 'limón'}
frutas
```

```
{'limón', 'manzana', 'naranja', 'sandía'}
```

CELL 02

```
vitamina_c = {'limón', 'perejil', 'fresa', 'naranja'}
```

CELL 03

```
vitamina_c
```

```
{'fresa', 'limón', 'naranja', 'perejil'}
```

CELL 04

```
frutas & vitamina_c
```

```
{'limón', 'naranja'}
```

CELL 05

```
frutas - vitamina_c
```

```
{'manzana', 'sandía'}
```

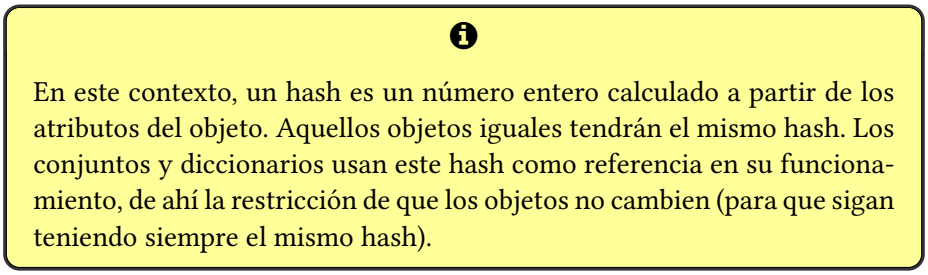
CELL 06

```
frutas | vitamina_c
```

```
{'fresa', 'limón', 'manzana', 'naranja', 'perejil', 'sandía'}
```

El conjunto es el tipo de dato dejado de lado más injustamente de todo el lenguaje, sospechamos que esto es así porque en otros lenguajes no se encuentra, entonces les programadores no se acostumbran a usarlo. Sin embargo, es muy poderoso y una vez que aprendamos a expresar nuestros algoritmos utilizando las capacidades de los conjuntos, evitaremos hacer un montón de bucles y comparaciones innecesarias en nuestros programas.

Sus elementos internos pueden ser solamente objetos inmutables (en verdad, podemos tener objetos que hagamos nosotros que siendo mutables, se les pueda calcular el `hash` en función de propiedades que no cambien, pero por simplicidad pensemos en objetos inmutables). Esto es porque para decidir si un elemento pertenece a un conjunto se usa su hash (de forma similar a los diccionarios que veremos luego). Una implicancia de esta propiedad es que es extremadamente rápido verificar si un elemento pertenece a un determinado conjunto, aunque este sea muy grande (porque se calcula su hash, y listo), en contraposición con lo que sucede con una lista o una tupla (porque en estos casos hay que comparar todos los objetos, algo potencialmente prohibitivo dependiendo del tamaño del contenedor).



En este contexto, un hash es un número entero calculado a partir de los atributos del objeto. Aquellos objetos iguales tendrán el mismo hash. Los conjuntos y diccionarios usan este hash como referencia en su funcionamiento, de ahí la restricción de que los objetos no cambien (para que sigan teniendo siempre el mismo hash).

Los conjuntos son objetos mutables, y como tales tienen métodos para agregar y eliminar elementos internos.

En el siguiente ejemplo vemos como podemos agregar objetos de a uno, o muchos simultáneamente (usando un iterable que los contiene). En ambos casos, tenemos que recordar la propiedad de los conjuntos de que no pueden tener objetos repetidos.

CELL 07

```
música = {'metal', 'tango', 'rock'}
música
```

```
{'metal', 'rock', 'tango'}
```

CELL 08

```
música.add('rock')
música
```

```
{'metal', 'rock', 'tango'}
```

Cell 09

```
música.add('jazz')
música
```

```
{'jazz', 'metal', 'rock', 'tango'}
```

Cell 10

```
música.update(['tango', 'cumbia'])
música
```

```
{'cumbia', 'jazz', 'metal', 'rock', 'tango'}
```

Cell 11

```
'rock' in música
```

```
True
```

Tenemos varias formas de remover elementos de los conjuntos, una que saca el elemento indicado pero falla si no está, otra que saca el elemento indicado (sin fallar si no está), y otra que saca un elemento de forma arbitraria (fallando si el conjunto estaba ya vacío):

Cell 12

```
música.remove('jazz')
música
```

```
{'cumbia', 'metal', 'rock', 'tango'}
```

Cell 13

```
música.remove('jazz')
```

```
KeyError                                  Traceback (most recent call last)
<ipython-input-13-533e3682df9b> in <module>
----> 1 música.remove('jazz')

KeyError: 'jazz'
```

Cell 14

```
música.discard('metal')
música
```

```
{'cumbia', 'rock', 'tango'}
```

Cell 15

```
música.discard('metal')
música
```

```
{'cumbia', 'rock', 'tango'}
```

Cell 16

```
música.pop()
```

```
'tango'
```

Cell 17

```
música
```

```
{'cumbia', 'rock'}
```

Como con el resto de los tipos, podemos usar el constructor para convertir entre ellos; en este caso el constructor toma cualquier iterable y se queda con los elementos que recibe:

Cell 18

```
set(['foo', 'bar'])
```

```
{'bar', 'foo'}
```

Cell 19

```
set("son ocho los orozco")
```

```
{' ', 'c', 'h', 'l', 'n', 'o', 'r', 's', 'z'}
```

1.3.7. Diccionarios

Los diccionarios también se delimitan con llaves, como los conjuntos, pero en este caso lo que se separan por comas son pares de objetos, cada par siendo una clave y un valor (separados entre sí por dos puntos).

Es que los diccionarios, a diferencia de los otros contenedores que vimos hasta ahora (listas, conjuntos, etc) guardan objetos identificados cada uno por una determinada clave. Los valores guardados pueden ser cualquier objeto de Python, mientras que las claves pueden ser objetos que se les puede calcular el hash.

La forma más directa de acceder a los valores guardados es a través de sus respectivas claves, pero también podemos listar todas las claves, todos los valores, e incluso todos los ítems (pares clave/valor).

Cell 01

```
d = {'foo': 23, 'bar': 90, 'baz': 133}
d
```

```
{'foo': 23, 'bar': 90, 'baz': 133}
```

Cell 02

```
d['foo']
```

```
23
```

Cell 03

```
d.keys()
```

```
dict_keys(['foo', 'bar', 'baz'])
```

Cell 04

```
d.values()
```

```
dict_values([23, 90, 133])
```

Cell 05

```
d.items()
```

```
dict_items([('foo', 23), ('bar', 90), ('baz', 133)])
```

Cada clave puede estar una sola vez en el diccionario (porque se accede a través del hash de las claves, de forma similar a lo que veíamos con los conjuntos), entonces si asignamos un nuevo valor a una clave ya presente, estaremos pisando el valor anterior. Si la clave no estaba, estaremos creando un nuevo ítem en el diccionario.

Cell 06

```
d
```

```
{'foo': 23, 'bar': 90, 'baz': 133}
```

Cell 07

```
d['foo'] = 1288
d
```

```
{'foo': 1288, 'bar': 90, 'baz': 133}
```

Cell 08

```
d['otra'] = 7
d
```

```
{'foo': 1288, 'bar': 90, 'baz': 133, 'otra': 7}
```

Tengamos en cuenta que desde Python 3.6 los diccionarios recuerdan el orden de inserción de sus claves (antes no tenían un orden en particular).

Si accedemos a una clave que no está presente en el diccionario, se genera una excepción de tipo `KeyError`. A veces es útil usar el método `get` que accede al diccionario y devuelve el valor correspondiente a la clave si es que la clave existe, pero en caso de que la clave no exista devolverá un valor que podemos especificar (o `None` si no indicamos nada).

Cell 09

```
d
```

```
{'foo': 1288, 'bar': 90, 'baz': 133, 'otra': 7}
```

Cell 10

```
d['foo']
```

```
1288
```

Cell 11

```
d['fux']
```

```
KeyError                                  Traceback (most recent call last)
<ipython-input-11-15405d883549> in <module>
----> 1 d['fux']

KeyError: 'fux'
```

Cell 12

```
d.get('foo')
```

```
1288
```

Cell 13

```
d.get('fux', "nada")
```

```
'nada'
```

Cell 14

```
d.get('fux')
```

Podemos borrar elementos del diccionario con la declaración **`del`**, pero también a veces es útil extraer el valor correspondiente a la clave que estamos borrando, y para eso tenemos el método `pop`, al que si le pasamos un valor por default soportará no encontrar a la clave indicada.

Cell 15

```
d
```

```
{'foo': 1288, 'bar': 90, 'baz': 133, 'otra': 7}
```

Cell 16

```
del d['bar']
d
```

```
{'foo': 1288, 'baz': 133, 'otra': 7}
```

Cell 17

```
d.pop('foo')
```

```
1288
```

Cell 18

```
d
```

```
{'baz': 133, 'otra': 7}
```

Cell 19

```
d.pop('foo')
```

```
KeyError                                  Traceback (most recent call last)
<ipython-input-19-9d43697015c6> in <module>
----> 1 d.pop('foo')

KeyError: 'foo'
```

Cell 20

```
d.pop('foo', "no estaba")
```

```
'no estaba'
```

Podemos crear diccionarios a partir de otras estructuras, cualquier iterable en verdad, pero tendrán que ser de pares de valores, donde cada par será clave/valor. También podemos crear

diccionarios tomando como fuente las claves, usando el método `fromkeys`, pero en este caso las claves tendrán siempre el mismo valor inicial. Y como detalle especial con este tipo de datos podemos usar una forma de pasar parámetros a las funciones que en este caso es particularmente útil (aunque sólo para el caso donde las claves son cadenas).

CELL 21

```
dict([('a', [1, 2]), ('b', [3, 4])])
```

```
{'a': [1, 2], 'b': [3, 4]}
```

CELL 22

```
dict.fromkeys(['foo', 'bar'])
```

```
{'foo': None, 'bar': None}
```

CELL 23

```
dict.fromkeys(['foo', 'bar'], 0)
```

```
{'foo': 0, 'bar': 0}
```

CELL 24

```
dict(a=[1, 2], b=[3, 4])
```

```
{'a': [1, 2], 'b': [3, 4]}
```

1.3.8. Iteradores

Antes de irnos de la Sección de Tipos de datos, mencionemos un concepto importante en Python: los iteradores.

A nivel de definición general, un iterador se refiere al objeto que permite al programador recorrer un contenedor. Prestemos atención a la diferencia entre el contenedor en sí (como colección de elementos) del iterador (que nos permite recorrer esos elementos).

Esta característica se expresa en muchos rincones del lenguaje. Tenemos la declaración **`for`**, que veremos más adelante en la Sección 1.4.3, que nos permite construir un bucle alrededor de iterar un objeto, pero también podemos hacerlo a mano, aunque de esta manera tenemos que pedirle al objeto iterable que nos de un iterador:

CELL 01

```
números = [1, 2, 3]
type(números)
```

```
list
```

CELL 02

```
iterador = iter(números)
type(iterador)
```

```
list_iterator
```

CELL 03

```
next(iterador)
```

```
1
```

CELL 04

```
next(iterador)
```

```
2
```

CELL 05

```
next(iterador)
```

```
3
```

CELL 06

```
next(iterador)
```

```
StopIteration                             Traceback (most recent call last)
<ipython-input-6-2389250a88e0> in <module>
----> 1 next(iterador)

StopIteration:
```

Esa excepción que vemos ahí es perfectamente normal, es el mecanismo que tienen los iteradores para indicar que no hay más elementos para entregar.

Una forma útil en el intérprete interactivo de iterar un objeto y ver el resultado es a través del constructor `list`. Usémoslo y veamos como en Python muchos tipos integrados en el lenguaje soportan el protocolo de iteración (decimos que son "iterables"):

CELL 07

```
list([1, 2, 3])  # una lista
```

```
[1, 2, 3]
```

CELL 08

```
list((1, 2, 3))  # una tupla
```

```
[1, 2, 3]
```

CELL 09

```
list("abcd")  # una cadena
```

```
['a', 'b', 'c', 'd']
```

CELL 10

```
list({1, 2, 3})  # un conjunto
```

```
[1, 2, 3]
```

Cell 11

```
list({'a': 1, 'b': 2, 'c': 3})  # un diccionario
```

```
['a', 'b', 'c']
```

No todos los tipos soportan este protocolo, claro. Por ejemplo, no se puede iterar un número entero, ya que no es un contenedor, no tiene elementos para entregar.

También tengamos en cuenta que normalmente cada vez que pidamos un iterador vamos a tener uno "fresco", que arrancará desde el principio (aunque nosotros podríamos cambiar este comportamiento si hacemos nuestros propios tipos de datos). Pero si trabajamos sobre el iterador puntualmente, podemos pedirle elementos de diversas maneras sin tener que volver a comenzar.

Cell 12

```
lista = [1, 2, 3, 4, 5, 6, 7]
list(lista)
```

```
[1, 2, 3, 4, 5, 6, 7]
```

Cell 13

```
iterador = iter(lista)
iterador
```

```
<list_iterator at 0x7f4a0cd2b700>
```

Cell 14

```
next(iterador)
```

```
1
```

Cell 15

```
next(iterador)
```

```
2
```

Cell 16

```
list(iterador)
```

```
[3, 4, 5, 6, 7]
```

Finalmente, prestemos atención cuando estamos iterando un objeto, ya que el mismo puede potencialmente entregar una cantidad muy grande de elementos, lo cual no tiene sentido tener en memoria simultáneamente. Mostremos un ejemplo de esto con la función integrada `range`, que entrega números en un determinado rango, ahí vemos como para un rango chico podemos iterarlo completamente con una lista, pero para uno muy grande, aunque podamos iterarlo a mano, realmente no tiene sentido hacerlo hasta el final.

Cell 17

```
nros = range(3)
list(nros)
```

```
[0, 1, 2]
```

Cell 18

```
muchos_nros = range(1000000000000000)
iter_muchos_nros = iter(muchos_nros)
next(iter_muchos_nros)
```

```
0
```

Cell 19

```
next(iter_muchos_nros)
```

```
1
```

Así y todo `range` en algún momento va a terminar. Pero podemos tener generadores que sean infinitos, ya veremos como construirlos cuando hablemos de Funciones 1.5.1.

1.4. Controles de flujo

En esta sección mostraremos las distintas declaraciones de Python que nos permiten controlar el flujo de ejecución de un programa, que hasta ahora veíamos como lineal.

Python tiene pocas estructuras en este sentido (no repitiendo innecesariamente funcionalidad, "simple es mejor que complejo"), pero las pocas que tiene son poderosas e interesantes.

En esta sección también incluimos *excepciones*, que aunque también se utilizan para manejar errores, en realidad es un concepto más amplio (se utilizan para situaciones excepcionales, no sólo errores), y la naturaleza de las mismas hacen que el flujo del programa se modifique.

1.4.1. If, elif, else

Comenzamos con el control de flujo más sencillo, el **`if`**. La estructura básica es muy simple:

```
if <expresión>:
    <bloque de código>
```

Si la expresión es verdadera, se ejecuta el bloque de código; si no, no.

¿Qué es una "expresión" en este contexto? Una expresión es una combinación de valores, variables, operadores y llamadas a funciones. Cuando sea evaluada, para el **`if`** va a terminar siendo falsa o verdadera, no hay otra.

Más abajo seguiremos hablando de las expresiones, pero ahora centrémonos en el bloque de código que es lo que se ejecuta si la expresión es verdadera. No es más que una secuencia de líneas de código, que puede ser una o más (no cero, y sin límites prácticos en su cantidad). El comienzo y finalización del bloque está marcado por su sangría (que muchas veces llamamos "indentación", término que aunque es usado muchísimo es solamente un anglicismo de *indentation*).

Esa sangría puede ser cualquier cantidad de espacios o tabuladores (¡pero no mezclarlos! y se recomienda usar cuatro espacios), con la condición de que sea consistente. Por ejemplo, si un bloque arranca con la linea cuatro espacios a la derecha, siempre estará sangrado igual durante todo el bloque (a menos que haya bloques anidados, claro) y luego terminará volviendo a la columna original.

Ejemplo:

```
1 if foo == 42:
2     print("foo vale 42")  # arranca el bloque con 4 a la derecha
```

```
    print("segunda linea")  # segunda linea ok
      print("tercera")  # esta está mal, porque está demasiado a la derecha
    if bar == 33:
        print("bar es 33")  # esta está bien, es un bloque nuevo adentro del otro
    print("muchas líneas")  # ok también: cierra el bloque interior, y vuelve a la columna original
  print("vamos cerrando")  # mal! si cerramos el bloque debemos volver a la columna original de este
print("afuera")  # esta está bien, ya afuera del bloque del if
```

A la estructura del **if** le podemos agregar una especie de continuación:

```
if <expresión 1>:
    <bloque de código 1>
elif <expresión 2>:
    <bloque de código 2>
```

Si la expresión 1 es verdadera, se ejecutará el bloque de código 1 y se ignorará el resto. Pero si la expresión 1 es falsa, se evalúa la expresión 2: si es verdadera se ejecutará el bloque 2, sino termina.

Y finalmente podemos agregarle una especie de salida final:

```
if <expresión 1>:
    <bloque de código 1>
elif <expresión 2>:
    <bloque de código 2>
else:
    <bloque de código 3>
```

El bloque de código 3 se ejecutará si todas las expresiones de la estructura fueron evaluadas a falso.

El **if** es obviamente obligatorio (arranca la estructura), pero el **elif** es opcional (y se pueden poner cuantos queramos, uno abajo del otro cada uno con su expresión a evaluar), y el **else** es también opcional pero no puede haber más de uno.

Veamos un ejemplo más real:

Cell 01

```
from datetime import date

hoy = date.today()
```

Cell 02

```
if hoy >= date(2030, 1, 1):
    print("El futuro ya llegó")
elif date(2020, 1, 1) <= hoy < date(2030, 1, 1):
    print('La famosa "década del 20"')
else:
    print("El lejano pasado")
```

```
La famosa "década del 20"
```

Claro que esa estructura se puede extender mucho más. En la actualidad Python no tiene una declaración "case" como muchos otros lenguajes. Esto se resuelve con estructuras **if/elif** si son relativamente pocos casos, o guardando funciones en un diccionario si son más.

En 2021, en la versión 3.10, Python ganó la posibilidad de realizar *pattern matching*, una característica bastante útil con muchas vueltas interesantes. Con esta nueva funcionalidad, usándola

de forma más bien básica, se podría tener algo similar a la declaración "case", pero les recomendamos mirar su PEP [14] para profundizar sobre este tema.

Volvamos sobre algo que prudentemente esquivamos al principio de la subsección: las expresiones.

Decíamos que una expresión es una combinación de valores, variables, operadores y llamadas a funciones. Esto es bastante genérico, y hay pocas cosas que no pueden ser incluidas en una expresión, como definiciones de funciones o clases, importar módulos, etc. En la práctica podemos hacer casi todo lo que deseamos, y esto nos permite ser bastante expresivos, por ejemplo como veíamos arriba al comparar un valor con el resultado de la función `date` que estamos llamando ahí mismo.

A esto hay que sumarle que todos los tipos de datos integrados en el lenguaje también son evaluables a verdadero o falso, lo cual podemos revisar sencillamente con la función `bool` (como regla general, el objeto evalúa a False si vale cero o está vacío, sino a True).

Cell 03

```
bool(-1)
```

```
True
```

Cell 04

```
bool([])
```

```
False
```

Cell 05

```
bool(None)
```

```
False
```

Cell 06

```
bool("")
```

```
False
```

Cell 07

```
bool("falso")
```

```
True
```

Y además tenemos los operadores de comparación:

- `==` igual a
- `!=` diferente de
- `is` es el mismo objeto que (identidad)
- `<` menor que
- `<=` menor o igual que
- `>` mayor que
- `>=` mayor o igual que

Todo esto nos permite ser muy expresivos al armar las estructuras `if`.

1.4.2. While

El **while** es una estructura de control de flujo que nos arma un bucle alrededor de ese bloque de código, repitiendo ese bloque en función de la evaluación de una expresión:

```
while <expresión>:
    <bloque de código>
```

Al arrancar el bucle, Python evalúa la expresión, si es falsa sale y nunca ejecuta el bloque de código. Si es verdadera ejecuta ese bloque de código, y vuelve a evaluar la expresión, si es falsa sale, sino ejecuta el bloque, y así hasta que la expresión de falso o se interrumpa por algo (pero potencialmente durante mucho mucho tiempo, hasta que el Sol se apague, digamos).

Veamos un ejemplo:

CELL 01

```
a = 0
while a < 3:
    a += 1
    print(a)
```

```
1
2
3
```

Ya hablamos al explicar el **if** 1.4.1 tanto de la expresión como del bloque de código, no hay mucho para agregar en ese aspecto. Por otro lado, con el **while** vemos que aparecen tres nuevas declaraciones, veámoslas.

El **break** nos permite interrumpir el bucle en la mitad de su ejecución. Si el código toca un **break**, entonces, el bucle se corta y sigue con lo que venía a continuación del mismo, sin terminar el bucle y sin volver a evaluar la expresión.

CELL 02

```
a = 0
while a < 5:
    a += 1
    if a == 3:
        break
    print(a)
```

```
1
2
```

El **continue** nos permite abortar la pasada actual del bucle, volviendo al principio del mismo, lo que incluye volver a evaluar la expresión. Vemos a continuación como el "3" no se imprime, porque al tocar el **continue** vuelve a recomenzar el bucle, sin llegar al **print** en ese caso.

CELL 03

```
a = 0
while a < 5:
    a += 1
    if a == 3:
        continue
    print(a)
```

```
1
2
4
5
```

Si usamos el **else** vamos a poder decidir si el bucle **while** terminó porque su expresión evaluó a falso o fue cortado con un **break**. Lo podemos pensar como en el **if**: si la expresión evalúa a verdadero se ejecuta el bucle, si evalúa a falso se ejecuta el bloque del **else**.

La combinación **while** con el **else** no es muy utilizado, en parte porque no lo vemos en otros lenguajes, pero es muy útil cuando justamente queremos saber si salimos del while en un caso o en el otro (lo que se resuelve en otros lenguajes utilizando otra variable como bandera).

Veamos ambos comportamientos en los siguientes ejemplos:

CELL 04

```
a = 0
while a < 2:
    a += 1
    if a == 3:
        break  # realmente no va a llegar acá
    print(a)
else:
    print("en el else")
print("afuera")
```

```
1
2
en el else
afuera
```

CELL 05

```
a = 0
while a < 4:  # ahora sí vamos a tocar el break
    a += 1
    if a == 3:
        break
    print(a)
else:
    print("en el else")
print("afuera")
```

```
1
2
afuera
```

1.4.3. For

El **for** es una declaración que nos permite recorrer iterables, ejecutando un bloque de código por cada uno de esos iterables (y haciendo referencia al elemento obtenido del iterable en cada momento usando un nombre que nosotros especificamos).

Si entendemos esa definición, la estructura es casi autodescriptiva:

```
for <nombre> in <iterable>:
    <bloque de código>
```

Veamos un ejemplo sencillo:

CELL 01

```
nros = [1, 2, 3]
for n in nros:
    print(n)
```

```
1
2
3
```

En realidad en lugar del "nombre" podemos tener varios nombres, si es que los elementos del iterable que recorremos pueden desempacarse correctamente, y obtendremos el mismo efecto que en la asignación múltiple (más específicamente, tenemos toda la experiencia del desempaquetado de tuplas que explicamos antes 1.3.4):

CELL 02

```
cosas = [('a', 1), ('b', 2), ('c', 3)]
for letra, nro in cosas:
    print(letra, nro ** 2)
```

```
a 1
b 4
c 9
```

Vale la pena que aclaremos que el `for` es más parecido al "foreach" de otros lenguajes, y diferente al "for" de C o al "do" de Fortran, por ejemplo, que sólo cuentan números entre un principio y final (en general para indizar una estructura y obtener los elementos internos de la misma). En verdad la necesidad de trabajar con rangos de números es real, y para eso Python tiene una función integrada `range`, en el que podemos especificar el final del rango (arrancando por default en cero), o inicio y final, e incluso el paso:

CELL 03

```
list(range(5))
```

```
[0, 1, 2, 3, 4]
```

CELL 04

```
list(range(3, 8))
```

```
[3, 4, 5, 6, 7]
```

CELL 05

```
list(range(3, 8, 2))
```

```
[3, 5, 7]
```

CELL 06

```
list(range(3, 12, 2))
```

```
[3, 5, 7, 9, 11]
```

Prestemos atención que el "desde" es inclusivo, mientras que el "hasta" es exclusivo; esto aunque quizás sea sorprendente tiene sentido en la foto más grande del funcionamiento general de Python. Por ejemplo, si queremos los índices para una lista de 4 elementos, haremos `range(4)` y eso nos dará el 0, 1, 2 y 3 que son las posiciones de una lista de 4 elementos. También es muy práctica la propiedad de que si hacemos `range(M, N)` la cantidad de números que obtendremos es `N - M`.

Para el ejemplo utilizamos el `list` porque el `range` es un generador de números, entonces tenemos que consumirlo para ver esos números. Por supuesto que lo podemos iterar directamente con el **`for`**:

CELL 07

```
for n in range(2, 5):
    print("El cuadrado de {} es {}".format(n, n ** 2))
```

```
El cuadrado de 2 es 4
El cuadrado de 3 es 9
El cuadrado de 4 es 16
```

El **`for`** es un bucle, y como con el otro bucle de Python (el **`while`**, de la Sección 1.4.2) podemos afectar el comportamiento utilizando el **`break`** (para interrumpir el bucle y salir), el **`continue`** (para abortar la pasada del bucle y volver al principio, obteniendo un nuevo elemento del iterable), e incluso el **`else`** (para discernir si el **`for`** terminó porque se nos acabó el iterable o porque cortamos con un **`break`**.

Veamos algunos ejemplos usando estas funcionalidades. Arrancamos con el **`break`** y el **`continue`**, viendo como corta en un caso y como vuelve al principio en el otro esquivando el resto del bucle:

CELL 08

```
números = [1, 2, 3, 4]
for n in números:
    if n == 3:
        break
    print(n)
```

```
1
2
```

CELL 09

```
for n in números:
    if n == 3:
        continue
    print(n)
```

```
1
2
4
```

Y veamos el **`else`**, para el caso en que terminamos el **`for`** por consumir totalmente el iterable o porque encontramos un **`break`**:

Cell 10

```
for n in números:
    print(n)
else:
    print("se nos terminó el iterable")
```

```
1
2
3
4
se nos terminó el iterable
```

Cell 11

```
for n in números:
    if n == 2:
        break
    print(n)
else:
    print("se nos terminó el iterable")
```

```
1
```

Hay un caso de uso típico en los `for` que es arrancar con una lista, realizarle una operación, y terminar con otra lista con los resultados de esa operación. Por ejemplo, podemos tener una lista de números y queremos calcular sus cuadrados:

Cell 12

```
números = [1, 3, -2, 2, 0, 5]
cuadrados = []
for n in números:
    cuadrados.append(n ** 2)

cuadrados
```

```
[1, 9, 4, 4, 0, 25]
```

Esta construcción es tan usual que Python tiene una sintaxis especial que nos permite escribir lo mismo pero de forma más reducida, con la ventaja que hasta queda más legible; se denomina *comprensión de listas* (en inglés *list comprehension*), y se define usando corchetes para delimitar la estructura, con los elementos sintácticos del `for` adentro:

Cell 13

```
números = [1, 3, -2, 2, 0, 5]
cuadrados = [n ** 2 for n in números]
cuadrados
```

```
[1, 9, 4, 4, 0, 25]
```

Podemos leer la segunda linea como "armamos una lista con ene al cuadrado para cada ene en números", y hace exactamente eso. Es mucho más fácil de entender que el `for` del ejemplo anterior que hace lo mismo pero a lo largo de varias líneas (entonces lo tenemos que seguir, ir y volver con la vista, entender qué hace con la lista que definimos al principio, etc.); en el caso de la *comprensión de listas* al primer vistazo (cuando reconocemos la estructura) ya sabemos que generamos una nueva lista realizando una operación con los elementos de un iterable, y nada más.

En realidad podemos complejizar apenas esa estructura, para el caso en que queramos filtrar

algunos elementos del iterable fuente. Veamos los mismos ejemplos que recién pero calculando logaritmos solamente para los valores mayores a cero.

CELL 14

```
import math
números = [1, 3, -2, 2, 0, 5]
logs1 = []
for n in números:
    if n > 0:
        logs1.append(math.log(n))

logs1
```

```
[0.0, 1.0986122886681098, 0.6931471805599453, 1.6094379124341003]
```

CELL 15

```
logs2 = [math.log(n) for n in números if n > 0]
logs2
```

```
[0.0, 1.0986122886681098, 0.6931471805599453, 1.6094379124341003]
```

Una generalización de esta estructura nos permite armar una *comprensión de conjuntos* (al delimitar la estructura con llaves) e incluso una *comprensión de diccionarios* (al delimitar la estructura con llaves y tener clave y valor separados por dos puntos:

CELL 16

```
números = [1, 3, -2, 2, 0, 5]
[n ** 2 for n in números]
```

```
[1, 9, 4, 4, 0, 25]
```

CELL 17

```
{n ** 2 for n in números}
```

```
{0, 1, 4, 9, 25}
```

CELL 18

```
{n: n ** 2 for n in números}
```

```
{1: 1, 3: 9, -2: 4, 2: 4, 0: 0, 5: 25}
```

1.4.4. Excepciones

El sistema de manejo de errores de Python es a través de excepciones.

1.4.4.1. ¿Qué son las excepciones?

Una excepción es un evento que ocurre durante la ejecución normal del programa e interrumpe el flujo normal del programa. En general, cuando un programa en Python encuentra una situación que no puede manejar, levanta una excepción.

Entonces, a diferencia de otros lenguajes que luego de llamar a una función (por ejemplo) tenemos que revisar el resultado para ver si indica que hubo un problema, en Python el resultado será lo que tenga que devolver la función normalmente, si es que la función terminó sin incon-

venientes. Pero si hubo un problema, obtendremos una excepción, que podemos manejar o dejar continuar.

Hagamos un pequeño programa para ver eso:

```
1 print("antes")
2 1 / 0
3 print("después")
```

Al ejecutar ese código, obtendremos:

```
$ python3 excep.py
antes
Traceback (most recent call last):
  File "x.py", line 2, in <module>
    1 / 0
ZeroDivisionError: division by zero
```

Vemos que se imprime el "antes", pero luego la ejecución se interrumpió al encontrar un error (al intentar dividir por cero). En ese punto, se levantó una excepción, y como no se capturó la terminó agarrando el intérprete de Python, el que interrumpió al programa y mostró un *traceback* por pantalla. Un traceback (término que usamos en inglés, ya que nunca se popularizó decirles "trazas de rastreo") es información que nos da el intérprete para entender de dónde viene el problema.

Tiene tres partes, un título que nos indica el comienzo del traceback (linea 3), un contenido cuyo largo dependerá de cuan profundo en la pila de llamadas a funciones haya sucedido el problema (líneas 4 y 5), y una última linea (la 6) mostrando el tipo de excepción y un mensaje (en nuestro caso `ZeroDivisionError`, y el mensaje indicando eso).

Para entender mejor la parte del medio veamos un ejemplo apenas más complejo (usando funciones, aunque nos adelantemos un poco a cuando las expliquemos formalmente en la Sección 1.5.1).

Veamos primero que pasaría si todo sale bien y no nos encontramos con ningún problema:

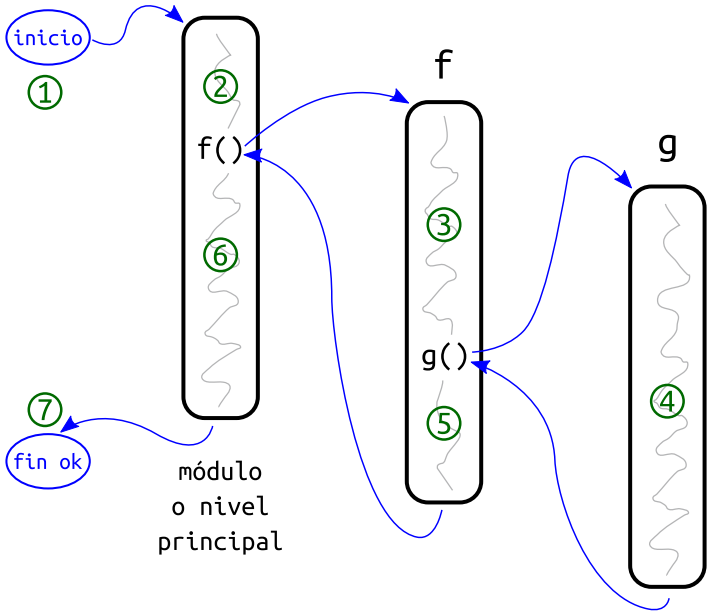
```
1 print("1. inicio")
2
3 def g():
4     print("4. en g")
5
6 def f():
7     print("3. antes en f")
8     g()
9     print("5. después en f")
10
11 print("2. antes en el módulo")
12 f()
13 print("6. después en el módulo")
14
15 print("7. fin")
```

Al ejecutar ese código, obtendremos:

```
1. inicio
2. antes en el módulo
```

```
3. antes en f
4. en g
5. después en f
6. después en el módulo
7. fin
```

Los números se corresponden al siguiente diagrama donde vemos la secuencia de ejecución de las distintas funciones:



Ahora veamos qué pasaría si en la función "g" nos encontramos con un problema. Modifiquemos esa función:

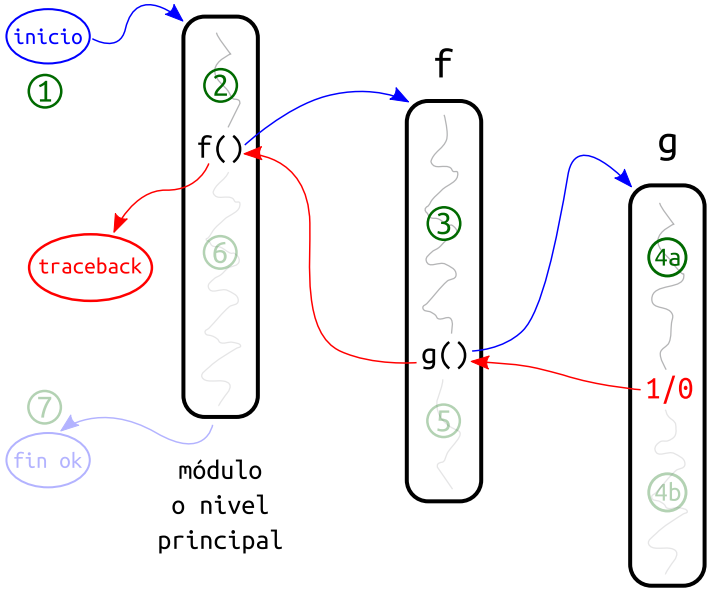
```
1  print("1. inicio")
2
3  def g():
4      print("4a. antes en g")
5      print("¿uno sobre cero?", 1 / 0)
6      print("4b. después en g")
7
8  def f():
9      print("3. antes en f")
10     g()
11     print("5. después en f")
12
13 print("2. antes en el módulo")
14 f()
15 print("6. después en el módulo")
16
17 print("7. fin")
```

Vemos que tenemos los resultados de los primeros prints y luego un traceback, pero no lo que veíamos antes:

```
1. inicio
2. antes en el módulo
```

```
3. antes en f
4a. antes en g
Traceback (most recent call last):
  File "x.py", line 14, in <module>
    f()
  File "x.py", line 10, in f
    g()
  File "x.py", line 5, in g
    print("¿uno sobre cero?", 1 / 0)
ZeroDivisionError: division by zero
```

Podemos ver mejor el flujo de ejecución en este otro diagrama que representa lo que acabamos de experimentar:



Allí vemos que el flujo de ejecución se modifica. Cuando llegamos al punto del problema, el resto de la función "g" no se ejecuta, sino que se genera una excepción que vuela hasta el punto donde la función fue llamada (linea roja que llega a "f"). El resto de esta función tampoco se ejecuta, como la excepción no fue capturada seguirá subiendo por las funciones hasta llegar a nivel de módulo, donde Python interrumpe el proceso mostrando el traceback.

La mejor forma que tenemos para entender lo que está pasando cuando hay un error es leer cuidadosamente el traceback. Recomendamos hacerlo de abajo para arriba.

Para el ejemplo anterior, vemos que tuvimos una excepción por división por cero en la linea 5 en la función "g" (donde hicimos `1 / 0`), lo cual viene de la linea 10 en la función "f" (donde hicimos `g()`), que a su vez viene de la linea 14 a nivel del módulo (donde hicimos `f()`).

Como tenemos un stack de largo tres (el cuerpo principal del programa más las dos funciones) tenemos tres pares de líneas en el centro del traceback.

Entonces, cada par de líneas de "la parte central" del traceback corresponde a un nivel del stack, y en cada caso nos muestra dónde se sucedió el problema para ese nivel del stack (en qué archivo, en qué linea y el contexto), y luego la linea en cuestión (que podríamos ver en el archivo/posición indicado, pero así es más cómodo).

1.4.4.2. Capturando y levantando excepciones

Podemos capturar las excepciones que se sucedan con un bloque **try**. La construcción típica es usarlo en conjunto al **except**:

```
try:
    <bloque de código>
except:
    <bloque de código>
```

El primer bloque de código, correspondiente al **try** es el supervisado, si se sucede alguna excepción en ese bloque, el **except** entra en juego y se ejecutará el segundo bloque de código; si ninguna excepción se levanta en el bloque supervisado, el **except** será ignorado completamente.

Veamos ambas situaciones en un ejemplo mínimo:

Cell 01

```
try:
    print("uno sobre dos:", 1 / 2)
except:
    print("hubo un problema")
```

Cell 02

```
try:
    print("uno sobre cero:", 1 / 0)
except:
    print("hubo un problema")
```

```
hubo un problema
```

Hay dos reglas de oro para seguir cuando escribimos estas estructuras. La primera es que debemos supervisar el mínimo de código posible (minimizar la cantidad de código dentro del bloque del **try**), la segunda es que debemos capturar solamente las excepciones que estamos esperando que puedan suceder (especificar el **except** lo más posible, no como hasta ahora que está capturando todo).

La primer regla es fácil de entender, pero veamos un ejemplo de cómo es útil especificar el **except** lo más posible. Supongamos el siguiente código, donde obtenemos un valor (supongamos de una medición) y calculamos uno sobre eso; excepcionalmente podemos tener un cero como valor, pero eso sería que el instrumento está descalibrado, entonces lo informamos y listo:

Cell 03

```
valor = 3  # todo bien
try:
    print("uno sobre algo:", 1 / valor)
except:
    print("instrumento descalibrado")
```

```
uno sobre algo: 0.3333333333333333
```

CELL 04

```
valor = 0  # efectivamente descalibrado
try:
    print("uno sobre algo:", 1 / valor)
except:
    print("instrumento descalibrado")
```

```
instrumento descalibrado
```

Sin embargo, supongamos que tenemos un problema más serio, y por un error en nuestro programa terminamos teniendo otra cosa como valor, una cadena:

CELL 05

```
valor = "error"  # otro problema desconocido
try:
    print("uno sobre algo:", 1 / valor)
except:
    print("instrumento descalibrado")
```

```
instrumento descalibrado
```

El mensaje que estamos dando es totalmente equivocado. Tengamos en cuenta que mucha veces también en el bloque del **except** se toman acciones para corregir o paliar el problema, y si tomamos las acciones equivocadas podemos estar complicando aún más la situación.

Entonces, tenemos que ser lo más específicos posibles al capturar la excepción. Para el ejemplo que estamos viendo, nosotros sabemos que podemos llegar a tener una **ZeroDivisionError**, ¡entonces capturemos sólo eso! Si le especificamos un tipo de excepción al **except** (o más de uno, entre paréntesis), capturará la excepción y ejecutará el bloque de código sólo si la excepción es de ese tipo (o de algunos de los varios tipos que pusimos entre paréntesis):

CELL 06

```
valor = 0
try:
    print("uno sobre algo:", 1 / valor)
except ZeroDivisionError:
    print("instrumento descalibrado")
```

```
instrumento descalibrado
```

CELL 07

```
valor = "error"
try:
    print("uno sobre algo:", 1 / valor)
except ZeroDivisionError:
    print("instrumento descalibrado")
```

```
TypeError                                 Traceback (most recent call last)
<ipython-input-7-3bd734f41923> in <module>
      1 valor = "error"
      2 try:
----> 3     print("uno sobre algo:", 1 / valor)
      4 except ZeroDivisionError:
      5     print("instrumento descalibrado")

TypeError: unsupported operand type(s) for /: 'int' and 'str'
```

Entonces, en el primer caso tenemos el mensaje que esperábamos, mientras que en el segun-

do caso la excepción no es capturada, la termina agarrando Python y nos genera el traceback correspondiente, lo cual está perfectamente bien, porque es información útil para encontrar el error en nuestro programa.

Hay situaciones, sin embargo, en que igualmente queremos capturar todo lo que pueda llegar a suceder, lo cual se justifica en procesos que son muy largos y que no queremos que se interrumpan nunca, o siempre tienen que estar levantados, y necesitamos capturar todas las excepciones para poder informarlas y seguir trabajando.

Pero incluso en estas situaciones no es recomendable poner el **`except`** "pelado", ya que hay excepciones que son internas al funcionamiento de Python y no debemos capturarlas. En estos casos es muy útil que las excepciones en Python estén dispuestas en forma de árbol, y que especificando un tipo de excepción realmente estamos capturando las excepciones de ese tipo y todas las de su rama (este árbol lo podemos ver en la documentación [15].

Entonces, podemos capturar **`Exception`** (que como vemos en el árbol de excepciones incluye a casi todas menos tres muy específicas), y obtendremos el efecto deseado. Es más, al especificar el tipo de excepción, podemos incluso ponerle un nombre a la excepción que capturamos y manejarla en el bloque de código.

Veamos todo esto en el ejemplo que traíamos, aprovechando la flexibilidad del **`except`** que nos permite especificar varios luego de un **`try`** (el comportamiento en estos casos es parecido a la cadena/secuencia que teníamos con los **`if`**/**`elif`**: se va verificando la excepción en orden en todos los **`except`**, si la excepción es capturada por uno de ellos se ejecuta su bloque de código y deja de verificarse en el resto.

Cell 08

```
valor = 0
try:
    print("uno sobre algo:", 1 / valor)
except ZeroDivisionError:
    print("instrumento descalibrado")
except Exception as exc:
    print("problema desconocido!", repr(exc))
```

```
instrumento descalibrado
```

Cell 09

```
valor = "queloqué"
try:
    print("uno sobre algo:", 1 / valor)
except ZeroDivisionError:
    print("instrumento descalibrado")
except Exception as exc:
    print("problema desconocido!", repr(exc))
```

```
problema desconocido! TypeError("unsupported operand type(s) for /: 'int' and 'str'")
```

En verdad, son más las herramientas que podemos usar con el **`try`**, no sólo tenemos el **`except`**, sino también el **`else`** y el **`finally`**, los cuales se pueden usar en cualquier combinación. La estructura completa sería:

```
try:
    <bloque de código>
except:
    <bloque de código>
else:
    <bloque de código>
```

```
finally:
    <bloque de código>
```

Veamos un resumen mostrando las características de cada una de esas partes:

- **`try`**: da comienzo al manejo de posibles excepciones, supervisando un bloque de código; es obligatorio incluirlo (da comienzo a la estructura) y puede estar una sola vez.
- **`except`**: ejecuta un bloque de código si en el código supervisado se levantó una excepción, y esa excepción es del tipo definido en el **`except`** (o incluida en su rama del árbol, como veíamos arriba); puede haber muchos o ninguno, si hay varios la comprobación de la excepción se hace en orden y se ejecuta el bloque de código solamente de aquel que captura la excepción.
- **`else`**: ejecuta su bloque de código solamente si en el código supervisado *no* se levantó una excepción; es opcional y puede haber a lo sumo uno.
- **`finally`**: ejecuta su bloque de código *siempre*, no importa qué haya pasado en el código supervisado; es opcional y puede haber a lo sumo uno.

Además de todo el manejo que podemos hacer sobre excepciones que son levantadas en alguna parte del código o por alguna situación, tenemos la opción de levantar nosotres mismos las excepciones integradas de Python, o incluso crear nuevas excepciones para nuestros programas.

Para levantar una excepción sólo tenemos que usar la declaración **`raise`** y la excepción y el mensaje que queremos levantar:

Cell 10

```
valor = -3
if valor < 0:
    raise ValueError("Valor inválido, no puede ser negativo.")
```

```
ValueError                                Traceback (most recent call last)
<ipython-input-10-2d18b9ea9787> in <module>
      1 valor = -3
      2 if valor < 0:
----> 3     raise ValueError("Valor inválido, no puede ser negativo.")

ValueError: Valor inválido, no puede ser negativo.
```

Vemos en el ejemplo como el tipo de excepción y el mensaje usados son los mostrados en el traceback.

También podemos usar el **`raise`** sin especificar una excepción, pero solamente en el contexto de manejar alguna excepción que hayamos capturado: en este caso el **`raise`** lo que hará es "relevantar" la misma excepción.

Es muy útil para los casos donde capturamos una excepción pero solamente para manejar algunos casos y otros no. Por ejemplo, en las siguientes líneas capturamos un posible error que ignoramos en un caso y en otros no:

CELL 11

```
import os
to_remove = '/tmp/somefile.txt'
try:
    os.remove(to_remove)
except FileNotFoundError as err:
    if err.filename.startswith("/tmp/"):
        print("Ignoramos no haber encontrado un temporal")
    else:
        raise
```

```
Ignoramos no haber encontrado un temporal
```

CELL 12

```
to_remove = '/opt/somefile.txt'
try:
    os.remove(to_remove)
except FileNotFoundError as err:
    if err.filename.startswith("/tmp/"):
        print("Ignoramos no haber encontrado un temporal")
    else:
        raise
```

```
FileNotFoundError                         Traceback (most recent call last)
<ipython-input-12-53dbdaf6d26c> in <module>
      1 to_remove = '/opt/somefile.txt'
      2 try:
----> 3     os.remove(to_remove)
      4 except FileNotFoundError as err:
      5     if err.filename.startswith("/tmp/"):

FileNotFoundError: [Errno 2] No such file or directory: '/opt/somefile.txt'
```

Definir nuestros propios tipos de excepciones es muy simple, aunque para ello necesitamos la sintaxis de clases (que veremos más adelante 1.5.2). El único requisito es que tenemos que "heredar" una excepción integrada de Python; de nuevo, el concepto de "herencia" todavía no lo vimos, pero para nuestro propósito es lo que ponemos entre paréntesis en la definición, veamos un ejemplo simple:

CELL 13

```
class FueraDeRango(Exception):
    """El valor medido está fuera del rango permitido."""
```

Es importante elegir de qué excepción heredamos porque eso es lo que termina armando "el árbol de excepciones" que mencionábamos arriba cuando decíamos que `except` captura el tipo de excepción indicado y a toda su rama.

En el caso del ejemplo estamos heredando `Exception`, pero podríamos heredar alguna otra excepción cuya semántica esté más cerca de la excepción que estamos definiendo (para ello es interesante empaparse de los tipos de excepciones que trae Python [16]. Por ejemplo, en nuestro caso probablemente estaríamos mejor heredar de `ValueError`, ya que está relacionada con un valor obtenido.

Una vez definida, la usamos como cualquier otra excepción:

CELL 14

```
raise FueraDeRango("valor negativo")
```

```
FueraDeRango                              Traceback (most recent call last)
<ipython-input-14-5161da85a699> in <module>
----> 1 raise FueraDeRango("valor negativo")

FueraDeRango: valor negativo
```

1.5. Encapsulando código

Encapsular código es el acto que nos permite acomodar determinadas líneas de código en alguna estructura para poder reutilizarlas a conveniencia.

La forma más sencilla en Python de lograr esto son las funciones, que es lo primero que estudiaremos en esta sección. Luego mostraremos las clases, que nos permiten encapsular no sólo el código sino también los objetos sobre los cuales trabaja ese código (abriéndonos las puertas a la Programación Orientada a Objetos), y finalmente hablaremos de módulos y paquetes, que son capas superiores que nos permiten encapsular funciones y clases para usarlas de distintos programas.

1.5.1. Funciones

Como mencionábamos al principio de la sección, la función es la estructura más sencilla para encapsular código.

Nos permite escribir un bloque de código (con sus estructuras de control de flujo, con sus propios bloques de código, etc) de forma que después podremos ejecutar ese bloque de código "llamando" a la función desde cualquier lado.

La forma más sencilla de la estructura de una función es:

```
def <nombre>():
    <bloque de código>
```

Esa estructura, aunque funcional, no nos permite pasarle datos ni obtener un resultado. Pero nos sirve para empezar a familiarizarnos con las funciones. Tenemos entonces un "nombre" que es el que usaremos para identificar a la función, y un bloque de código (indentado, como corresponde).

Es importante entender la diferencia entre "definir" una función y "llamar a" (o "ejecutar") una función. En el primer caso solamente hacemos que Python compile la estructura y la tenga en memoria lista para usar, mientras que en el segundo caso es realmente cuando el bloque de código de la función se termina ejecutando.

En el siguiente ejemplo podemos ver primero la definición en sí de la función, y cómo podemos referenciarla con su nombre (en el `print`, o directamente en el intérprete interactivo, y la diferencia fundamental con ejecutar esa función, al final del ejemplo, cuando escribimos el nombre de la función seguida de paréntesis (sin nada entre ellos, en este caso, porque la función no recibe parámetros).

CELL 01

```
def foo():
    print(5)
```

Cell 02

```
print(foo)
```

```
<function foo at 0x7fc314631af0>
```

Cell 03

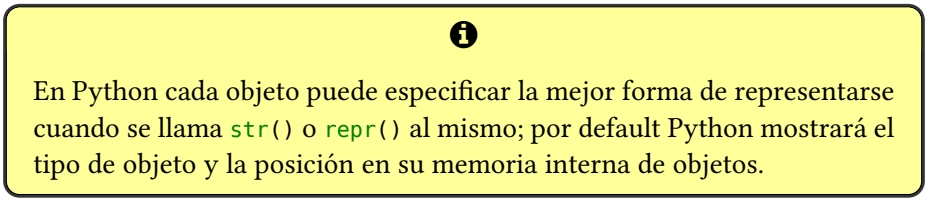
```
foo
```

```
<function __main__.foo()>
```

Cell 04

```
foo()
```

```
5
```



En Python cada objeto puede especificar la mejor forma de representarse cuando se llama `str()` o `repr()` al mismo; por default Python mostrará el tipo de objeto y la posición en su memoria interna de objetos.

Tener una función como esa es válido en algunos casos, pero en realidad la mayoría de las veces estaremos pasándole valores a la función y/o recibiendo resultados de la misma.

Para recibir valores, los tenemos que especificar en la definición de la función. Esto se logra de distintas maneras, y es bastante flexible (lo veremos más abajo), pero por ahora, simplificando, digamos que escribimos los nombres con los que haremos referencia a esos valores, y los podremos acceder desde el bloque de código de la función.

En el siguiente ejemplo definimos una función que recibe dos valores (sería un error pasarle uno o tres):

Cell 05

```
def foo(a, b):
    print(a, b)
```

Cell 06

```
foo
```

```
<function __main__.foo(a, b)>
```

Cell 07

```
foo(3, 4)
```

```
3 4
```

Hasta ahora la función ejecuta su bloque de código y termina. Por default, la función siempre devuelve `None` al terminar, pero tenemos control sobre eso mediante la declaración **`return`**.

Podemos poner cualquier cantidad de **`return`**s en una función. Si el flujo del código pasa por una linea con **`return`**, la función termina y devuelve lo que allí se indica (no importa si hay otros **`return`**s en otros lados de la función).

CELL 08

```
def foo(a, b):
    return a + b
```

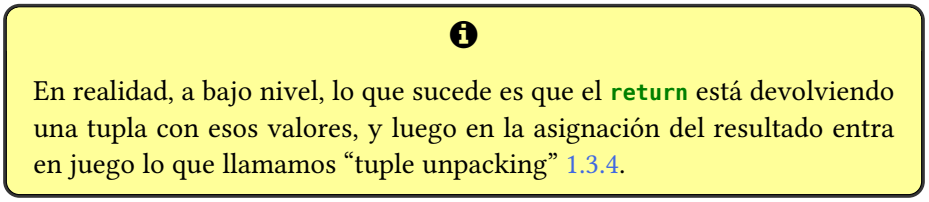
CELL 09

```
r = foo(3, 4)
r
```

```
7
```

Prestemos atención al detalle de haber llamado a la función y realizar una asignación con el resultado de esa función, para poder trabajar luego con el mismo.

Tengamos en cuenta que podemos obtener más de un resultado cuando termina una función, para lo cual el `return` soporta que escribamos diferentes valores separados por coma y podemos acceder a esos valores con una asignación múltiple.



En realidad, a bajo nivel, lo que sucede es que el `return` está devolviendo una tupla con esos valores, y luego en la asignación del resultado entra en juego lo que llamamos "tuple unpacking" 1.3.4.

Ya con un ejemplo más complejo, armemos una función que recibe dos valores y devuelve la multiplicación y la división de ambos números (¡si es posible! si el segundo número es cero devuelve None allí):

CELL 10

```
def foo(a, b):
    mul = a * b
    if b == 0:
        return mul, None

    div = a / b
    return mul, div
```

CELL 11

```
foo(3, -2)
```

```
(-6, -1.5)
```

CELL 12

```
m, d = foo(3, -2)
```

CELL 13

```
m
```

```
-6
```

CELL 14

```
d
```

```
-1.5
```

Cell 15

```
print("Resultados: multip={} divis={}".format(m, d))
```

```
Resultados: multip=-6 divis=-1.5
```

Cell 16

```
m, d = foo(3, 0)
print("Resultados: multip={} divis={}".format(m, d))
```

```
Resultados: multip=0 divis=None
```

No hay más para explorar por el lado de devolver valores, así que volvamos sobre el otro lado de usar funciones: pasarle parámetros.

Vayamos mostrando las distintas alternativas. El modo más básico es lo que veníamos haciendo, definir algunos parámetros y pasar valores para los mismos. Veamos cuando esto funciona bien, y también los errores que tenemos al no respetar ese "acuerdo básico":

Cell 17

```
def foo(a, b):
    print(a, b)
```

Cell 18

```
foo(5, 6)
```

```
5 6
```

Cell 19

```
foo(5)
```

```
TypeError                                 Traceback (most recent call last)
<ipython-input-19-c5b1cea4a11b> in <module>
----> 1 foo(5)

TypeError: foo() missing 1 required positional argument: 'b'
```

Cell 20

```
foo(5, 6, 7)
```

```
TypeError                                 Traceback (most recent call last)
<ipython-input-20-5b675c25d8d5> in <module>
----> 1 foo(5, 6, 7)

TypeError: foo() takes 2 positional arguments but 3 were given
```

Un detalle básico pero interesante es que hay una correspondencia ordinal entre los parámetros especificados en la definición de la función y los valores que pasamos al llamarla: el `5` va a la `a` y el `6` va a la `b`; es por esto que llamamos "posicionales" a estos argumentos (lo vemos también en el mensaje de error en el ejemplo).

Cuando definimos la función podemos especificar que algunos de esos parámetros tengan un valor por default, entonces no va a ser necesario pasarlos cuando llamemos a la función:

Cell 21
```
def foo(a, b=1, c=3):
    print(a, b, c)
```

Cell 22
```
foo(9)
```
```
9 1 3
```

Cell 23
```
foo(9, 5)
```
```
9 5 3
```

Cell 24
```
foo(9, 5, 7)
```
```
9 5 7
```

En el ejemplo vemos que si pasamos sólo el valor para `a`, `c` y `b` tienen sus valores por default. En la segunda llamada pasamos valor para `a` y para `b` (de nuevo, porque son posicionales, el primer valor al primer parámetro, etc.), pero no para `c`. Y finalmente, vemos que si les pasamos valores para los tres, no se consideran sus valores por default.

¿Pero cómo haríamos en el ejemplo anterior para pasar un valor a `a` y a `c`, pero no a `b` (y que tome su valor por default)? Para ello nos tendríamos que salir del esquema de parámetros posicionales y empezar a nombrarlos, lo cual es tan sencillo como especificar para qué parámetro queremos que vaya cada valor:

Cell 21
```
def foo(a, b=1, c=3):
    print(a, b, c)
```

Cell 25
```
foo(9, c=7)
```
```
9 1 7
```

En este caso vemos que pasamos un `9` que va a `a` (¡posicional!) pero luego especificamos que el `7` va para `c`; a `b` no le terminamos pasando un valor, así que toma su default.

En realidad una vez que nombramos los parámetros, podemos escaparnos totalmente al orden de sus posiciones, más allá que en la definición tengan valores por default o no:

Cell 21
```
def foo(a, b=1, c=3):
    print(a, b, c)
```

Cell 26
```
foo(9, c=7, b=8)
```
```
9 8 7
```

CELL 27

```
foo(c=7, a=33)
```

```
33 1 7
```

Hasta ahora estamos manejando cantidad de fija de parámetros. Python soporta que en la definición usemos el * que consumirá todos aquellos valores que pasemos por posición que no hayan sido tomados todavía:

CELL 28

```
def foo(a, b, *c):
    print(a, b, c)
```

CELL 29

```
foo(1, 2)
```

```
1 2 ()
```

CELL 30

```
foo(1, 2, 3, 4)
```

```
1 2 (3, 4)
```

En el ejemplo vemos como en la primer llamada los dos valores que pasamos van a los primeros dos parámetros definidos, pero no quedó nada para c (entonces es una tupla vacía), mientras que en el segundo caso "sobraron" dos valores, entonces c si tiene contenido.

También Python nos ofrece el **, que de manera similar consumirá todos los nombrados que no hayan encontrado otro lugar:

CELL 31

```
def foo(a, b, **c):
    print(a, b, c)
```

CELL 32

```
foo(a=1, b=2)
```

```
1 2 {}
```

CELL 33

```
foo(a=1, c=7, b=3, d=8)
```

```
1 3 {'c': 7, 'd': 8}
```

Para el caso de * la estructura donde Python guarda los argumentos posicionales sobrantes es una tupla, ya que sólo es importante el orden, mientras que para el ** como tenemos valores y nombres, la estructura útil para guardar eso es el diccionario.

Obviamente se pueden combinar todos los casos que estuvimos viendo hasta recién .

CELL 34

```
def foo(a, b, *c, d=7, e=8, **f):
    print(a, b, c, d, e, f)
```

CELL 35

```
foo(1, 2)
```

```
1 2 () 7 8 {}
```

CELL 36

```
foo(1, 2, 3, 4, e=9, j=15)
```

```
1 2 (3, 4) 7 9 {'j': 15}
```

Todo esto funciona mientras no haya ambigüedades en la definición o en el llamado a la función; en esos casos Python mostrará un mensaje de error indicando el problema.

CELL 37

```
def foo(a=7, b):
    print(a, b)
```

```
  File "<ipython-input-37-258463965655>", line 1
    def foo(a=7, b):
            ^
SyntaxError: non-default argument follows default argument
```

CELL 38

```
def foo(a, b=7, **c):
    print(a, b, c)

foo(1, 3, a=5)
```

```
TypeError                                 Traceback (most recent call last)
<ipython-input-38-cf3d3e66d1d8> in <module>
      2     print(a, b, c)
      3
----> 4 foo(1, 3, a=5)

TypeError: foo() got multiple values for argument 'a'
```

Así como podemos usar en la definición el * para guardar excedentes posicionales en una tupla y ** para los excedentes nombrados en un diccionario, podemos usarlos en las llamadas a las funciones para "desarmar" una tupla con los valores o un diccionario con los nombres/valores:

CELL 39

```
def foo(a, b, c):
    print(a, b, c)
```

CELL 40

```
t = (1, 2, 3)
foo(*t)
```

```
1 2 3
```

Cell 41

```
d = {'b': 2, 'c': 3, 'a': 1}
foo(**d)
```

```
1 2 3
```

Cabe acotar que no estamos mencionando todos los casos posibles, y que hay más reglas y operadores (como forzar a que los parámetros sean nombrados o posicionales), y algunos detalles más, explicados en profundidad en la referencia del lenguaje **??**.

1.5.1.1. Espacios de nombres

Cuando explicamos cómo funcionaba Python con sus objetos y nombres (en lugar de variables con valores, ver Sección 1.3.5), usamos unos diagramas donde a la derecha teníamos los objetos en memoria, y a la izquierda otra zona donde poníamos los nombres. Este espacio reservado para los nombres se llama justamente "espacio de nombres" (en inglés "namespace"), y es una zona de memoria donde justamente se guardan los nombres que referencian a los otros objetos.

Traemos esto a colación en esta sección porque en Python no tenemos solamente un espacio de nombres, sino que pueden haber muchos, y las funciones tienen mucho que ver en eso.

Cuando arranca Python tenemos un espacio de nombre que se conserva hasta que el proceso termina y es accesible desde todos lados: el espacio de nombre "global". Por otro lado, cada vez que ejecutamos una función, se crea otro espacio de nombres, "local" a la función, que permanecerá activo mientras las función se está ejecutando y desaparecerá cuando la misma termine.

Tenemos que tener en cuenta que lo que se destruye al terminar la función es el espacio de nombre, no los objetos referenciados por los mismos. Claro, algunos objetos quedarán sin referencia luego de que el espacio de nombre desaparezca (y de esos se encarga la administración automática de memoria de Python), pero puede ser que otros objetos estén referenciados de otros lados, y sigan vivos.

Veamos un ejemplo sencillo:

Cell 42

```
def foo(a, b):
    x = a * b
    return x
```

Cell 43

```
x = 3
y = 5
foo(x, y)
```

```
15
```

Cell 44

```
x
```

```
3
```

Analicémoslo en detalle, por partes. Lo primero que tenemos es una definición de una función (que todavía no ejecutamos, claro). Luego le ponemos nombre a dos enteros (`x` e `y`), que usamos para llamar a la función. En *ese* momento se ejecuta la función, que nos devuelve 15.

Si vamos a la ejecución de la función, vemos que esos dos enteros los recibe en dos parámetros que llama `a` y `b`, realiza un cálculo y devuelve ese valor. Como parte del procesamiento, la función

también define un nombre `x`, pero este se define en el espacio de nombres *local* de la función (igual que `a` y `b`, para el caso).

La `x` definida en el espacio de nombres local apunta al entero 15, y luego de ejecutar la función vemos que, afuera, `x` sigue apuntando al 3 original. Esto es porque afuera estamos usando el espacio de nombres global, no se nos mezcla con el espacio de nombres local de la función.

Es importante entender la diferencia entre "definir" un nombre en un espacio de nombres, y tener "acceso" a ese nombre (también decimos "ver" ese nombre, en inglés se usa "scope"). Veamos el siguiente ejemplo para resaltar esta diferencia:

Cell 45

```
def foo():
    y = 2
    z = 3
    print(x, y, z)
```

Cell 46

```
x = 8
y = 9
foo()
```

```
8 2 3
```

Cell 47

```
x
```

```
8
```

Cell 48

```
y
```

```
9
```

Cell 49

```
z
```

```
NameError                                 Traceback (most recent call last)
<ipython-input-49-3a710d2a84f8> in <module>
----> 1 z

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

Arrancamos definiendo una función (que veremos en detalle a continuación, cuando la ejecutemos), y luego se definen en el espacio de nombres global una `x` apuntando a un 8 y una `y` apuntando a un 9.

Cuando ejecutamos la función, esta primero define una `y` apuntando a un 2 y una `z` apuntando a un 3 (en ambos casos, en el espacio de nombres local de la función), y luego hace un print de tres nombres: para `x` e `y` es simple, porque está mostrando lo que encuentra en el espacio de nombres local, pero `z` nos puede sorprender.

Es aquí donde tenemos que entender que a nivel de visibilidad, desde adentro de la función Python intenta resolver el nombre primero buscando en el espacio de nombres local, y luego si no la encuentra allí busca en el espacio de nombres global. Esta secuencia es importante, porque eso determina que el `y` que encuentra es el que apunta al 2 (¡no al 9!), y para `z` que no está local pero si global, igual la encuentra.

Por otro lado, desde afuera de la función no tenemos visibilidad a su espacio de nombres, por eso cuando al final queremos ver el valor de z nos da error de nombre, porque z no está definida en el espacio de nombres global (y es en el único en que busca).

Un detalle importante es que como desde adentro de la función tenemos visibilidad sobre los objetos del espacio de nombres global, si los objetos son mutables podremos modificarlos:

CELL 50

```
def foo():
    x.append(3)
```

CELL 51

```
x = [1]
x
```

```
[1]
```

CELL 52

```
foo()
x
```

```
[1, 3]
```

CELL 53

```
foo()
x
```

```
[1, 3, 3]
```

Aunque podría considerarse una mala práctica de programación (porque la función nos está cambiando objetos que viven fuera de ella), en algunos casos es útil y muy ventajoso poder hacer eso. ¡Usar con precaución!

1.5.1.2. Generadores

Los generadores son un tipo particular de objeto que cuando los iteramos nos dan elementos, pero no los tenían de antes. Los van generando en el momento. Un ejemplo integrado en Python es el `range`. Si hacemos `range(10 ** 100)` obtenemos un objeto que si le pedimos, nos dará enteros entre 0 y 10^{100} - 1, pero obviamente no preparó todos esos números en el momento. Los irá *generando*.

Se usan mucho en Python porque optimizan el uso de memoria y mejoran el rendimiento general. Veamos el siguiente ejemplo que aunque muy simple, muestra una optimización clara que repetida por varios rincones del lenguaje hacen una diferencia importante:

CELL 54

```
sum(range(10))
```

```
45
```

El ejemplo suma los números del 0 al 9, generados por el `range`. Si `range` (en vez de funcionar como generador) armara una lista con los números del 0 al 9, el efecto sería exactamente el mismo, con el detalle que como parte del proceso, se construyó una lista con todos los números en memoria simultáneamente, que luego fue consumida por el `sum`. Esa lista no sirvió para nada, en realidad, sólo ocupó memoria y tiempo para su creación/administración.

Hay distintas formas de construir generadores en Python, pero una de las más simples es armar una "función generadora". Parece una función normal con la excepción que dentro de su bloque de código usa la declaración **`yield`**, que justamente le cambia el comportamiento.

¿Recuerdan que dijimos que las funciones cuando las llamamos se ejecutan hasta que terminan, destruyendo su espacio de nombres local, y que cuando las volvemos a llamar vuelven a ejecutarse desde el principio? Bueno, justamente las funciones generadoras cambian ese comportamiento. Cuando llamamos a la función nos devuelve un generador. Cuando le vamos pidiendo elementos a ese generador lo que va a hacer es ejecutar esa función hasta que llega a un **`yield`**, devolviéndonos lo que allí se indica, "pausando" la ejecución de ese código. Y cuando le pidamos el próximo elemento al generador, ese código "se despertará" en el punto en que estaba y continuará su ejecución hasta que termine o encuentre otro **`yield`**.

Para ver esto en un ejemplo, primero hagamos una versión casera y recortada del `range` con una función clásica, para poder comparar ambos códigos.

Cell 55

```
def rango(limite):
    nros = []
    n = 0
    while n < limite:
        nros.append(n)
        n += 1
    return nros
```

Cell 56

```
rango(10)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Cell 57

```
sum(rango(10))
```

```
45
```

En esta función vemos lo que mencionábamos arriba. La función genera una lista con los números, luego el `sum` la consume sumando esos números y produce el 45 como resultado. No sólo la lista es innecesaria, también mantener todos los números al mismo tiempo en memoria sólo para sumarlos es un desperdicio de recursos. Esto es obviamente un factor si tenemos una lista muy grande, pero tampoco hay que desestimar la situación con estructuras pequeñas, porque cualquier necesidad de memoria puede disparar que el proceso que estamos ejecutando tenga que salir a pedirle memoria al sistema operativo, y eso siempre es caro.

Convirtamos la función de arriba en generadora.

Cell 58

```
def rango(limite):
    n = 0
    while n < limite:
        yield n
        n += 1
```

CELL 59

```
g = rango(10)
g
```

```
<generator object rango at 0x7fc314591d60>
```

CELL 60

```
next(g)
```

```
0
```

CELL 61

```
next(g)
```

```
1
```

CELL 62

```
sum(rango(10))
```

```
45
```

La estructura definida es muy similar, pero notemos que no tenemos la estructura interna "nros" (porque no estamos creando esa lista), y que aparece el famoso **`yield`**.

Cuando ejecutamos esta función generadora, realmente no se empieza a ejecutar el bloque de código, sino que obtenemos el objeto generador en sí, como mostramos en el ejemplo. Al hacer el primer `next` se empieza a ejecutar el código, hasta que llega al **`yield`**, allí devuelve el valor que teníamos en `n` (el `0`) y la ejecución del código queda suspendida (ya que el control lo tenemos nosotros en el intérprete interactivo). Cuando hacemos el segundo `next`, la ejecución no arranca desde el principio, sino que continua desde donde estaba, sumándole `1` a `n`, luego vuelve a evaluar la expresión del **`while`** y llega nuevamente al **`yield`**, devolviéndonos ahora el `1`.

Si siguiéramos pidiéndole números con el `next`, los seguiríamos obteniendo hasta que la expresión del **`while`** de falso, y en ese caso vemos que se termina la función. Como es una función generadora, cuando sale de la función en realidad se genera la excepción **`StopIteration`** que es la que usa Python para indicar que no hay más ítems para iterar. Esto nos permite integrar a estos generadores en todas las estructuras normales de Python.

Finalmente entonces, usamos en el `sum` la función generadora que armamos, que vemos que nos devuelve el mismo resultado que con la estructura clásica, pero sin construir la lista intermedia.

También podemos aplicar este concepto de "generador" a las comprensiones de listas 1.4.3, armando directamente "comprensiones generadoras", usando paréntesis en lugar de corchetes

CELL 63

```
(x ** 2 for x in range(3))
```

```
<generator object <genexpr> at 0x7fc314591e40>
```

Incluso podemos obviar esos paréntesis cuando tenemos a la expresión generadora dentro de una llamada a función (porque los dobles paréntesis son superfluos en Python).

Veamos un simple ejemplo donde sumamos los cuadrados del 0 al 9 de dos formas distintas.

Cell 64

```
sum([x ** 2 for x in range(10)])
```

```
285
```

Cell 65

```
sum(x ** 2 for x in range(10))
```

```
285
```

En ambos casos arrancamos con `range` (que es generadora), pero en el primero armamos una lista intermedia con los cuadrados (notemos los corchetes que arman la compresión de listas), mientras que en el segundo tenemos una comprensión generadora (no hace falta poner los paréntesis, ya que aprovechamos los de alrededor.

Para el `sum` es exactamente lo mismo, y tenemos el mismo resultado, pero el segundo caso es más rápido y hasta más legible y conciso.

1.5.2. Clases

Las *clases* son una forma de encapsular código junto a los objetos que son manejados por ese código.

Aunque suena simple, realmente esto nos permite separar en distintas estructuras los distintos objetos que necesitamos manejar con el código para procesarlos, lo cual nos permite modelar eficientemente la realidad que estamos tratando de representar, abriéndonos las puertas de la Programación Orientada a Objetos (en adelante "POO").

En este libro mostraremos el funcionamiento básico de las clases, la definición de su estructura, qué implica instanciarlas y las bases del funcionamiento de los objetos que obtenemos, pero no pretendemos enseñar POO en un capítulo, ya que es un tema para todo un libro (¡o más de uno!).

Entonces la idea es que podamos leer y entender código que usa clases, e incluso construir algunas sencillas, sin pretender desbloquear todo su potencial.

La estructura básica de una clase es extremadamente simple:

```
class <nombre>:
    <bloque de código>
```

Con eso ya armamos una clase con el nombre que queramos, y tenemos un bloque de código para continuar. Como con las funciones, las clases nos crean un espacio de nombres diferente, y allí es que empezaremos a agregar código que luego utilizaremos más adelante.

Armemos un ejemplo con más elementos, para empezar a explicarlos:

Cell 01

```
class Foo:
    a = 3
    def f():
        print(8)

Foo
```

```
__main__.Foo
```

CELL 02

```
Foo.a
```

```
3
```

CELL 03

```
Foo.f
```

```
<function __main__.Foo.f()>
```

CELL 04

```
Foo.f()
```

```
8
```

En el bloque de código definimos una variable y una función, y luego las usamos. Veamos que hicimos `Foo.a` y `Foo.f()`, porque tanto "a" como "f" no están en el espacio de nombres global, sino en el de la clase (y como con los módulos, cuando hacíamos `math.sqrt(2)` 1.2.3, usamos el punto para indicar que estamos usando un nombre de "adentro" de otro objeto). Entonces, es claro que tanto "a" como "f" están adentro de la clase.

Lo que es raro en ese ejemplo es como estamos usando la clase. En realidad el uso normal de una clase es para generar objetos, como es regla en la POO. Es más, como ya estamos metiéndonos en ese mundillo, adoptemos dos términos que son de uso genérico allí y que vamos a encontrar en muchos textos. Las variables dentro de las clases y objetos las llamaremos "atributos", mientras que las funciones que definimos allí adentro las llamaremos "métodos". Entonces, de acuerdo con esta nueva terminología, diremos que tenemos "una clase Foo con un atributo a y un método f".

Ahora hagamos el último salto y armemos una clase que tiene sentido ser usada para instanciar objetos.

CELL 05

```
import math

class TriánguloRectángulo:
    def __init__(self, cateto1, cateto2):
        self.cateto1 = cateto1
        self.cateto2 = cateto2
    def hipotenusa(self):
        return math.sqrt(self.cateto1 ** 2 + self.cateto2 ** 2)
```

Epa, ¡cuantas cosas nuevas! Vayamos entendiéndolas por partes, viendo cómo encaja en este uso distinto que mencionamos arriba.

Las clases, en el paradigma de POO, son las estructuras que encapsulan el código para procesar determinada información, junto a dicha información. Y funcionan como plantillas, que nos darán distintos objetos cada vez que *instanciemos* la clase. Estos objetos serán del mismo tipo (el tipo de los objetos es la clase en sí), y por lo tanto el código encapsulado será el mismo, aunque ese código procesará la distinta información que tendremos en cada objeto.

En nuestro ejemplo tenemos una clase TriánguloRectángulo, donde encapsulamos un código (cómo calcular la hipotenusa a partir de los catetos) junto a la información en sí (los catetos). Si queremos trabajar con distintos triángulos, obviamente tendremos una variedad de pares de catetos, pero en todos los casos la forma de calcular la hipotenusa es la misma; esto está en linea con la filosofía de la POO de que estos objetos modelan y representan nuestra realidad.

Armemos entonces dos triángulos, para experimentar y seguir entendiendo ese código (sin

repetir aquí la definición de la clase, por brevedad):

Cell 06

```
t1 = TriánguloRectángulo(4, 5)
t1
```

```
<__main__.TriánguloRectángulo at 0x7f18bc45f880>
```

Cell 07

```
t2 = TriánguloRectángulo(10, 1)
t2
```

```
<__main__.TriánguloRectángulo at 0x7f18bc45f910>
```

Cell 08

```
t1.cateto1
```

```
4
```

Cell 09

```
t2.cateto1
```

```
10
```

Vemos que al nombre de la clase le estamos agregando paréntesis, como hacemos con las funciones cuando las ejecutamos. Aquí es similar, pero a la clase la estamos *instanciando*, lo que nos devuelve un "objeto del tipo TriánguloRectángulo", que guardamos en t1 (y luego hacemos lo mismo con t2).

Cuando le decimos al intérprete interactivo que nos muestre esos objetos, vemos que nos dice que son del tipo TriánguloRectángulo y nos dice que están en posiciones de memoria distintas (con lo cual podemos deducir que son dos objetos distintos). Es más, cuando nos fijamos el valor de `cateto1` de ambos objetos vemos que obtenemos distintos valores (¡son distintos objetos!), cada uno teniendo el primer valor que pasamos cuando instanciamos la clase.

¿Cómo sucedió eso? Si volvemos a la definición de la clase, vemos que allí teníamos un método con un nombre especial, `__init__`. Este método se ejecuta automáticamente cuando instanciamos la clase donde está definido. Entonces, cuando hicimos `TriánguloRectángulo(4, 5)` se instanció la clase y se ejecutó ese método de inicialización, pasándole justamente estos valores que indicamos nosotros.

Los métodos especiales son un conjunto de métodos predefinidos, con comportamientos específicos definidos en el lenguaje mismo [17], que Python utiliza para interactuar con los objetos en todo nivel, por ejemplo llamando a `__init__` para inicializar una clase, o `__iter__` cuando recorremos un objeto con el for.

Pero si prestamos un poco más de atención veremos que en su definición `__init__` declara que tiene que recibir 3 parámetros (`self`, `cateto1` y `cateto2`), mientras que nosotros estamos pasando solamente dos. Es que para todo lo que es métodos en las clases, Python inserta automáticamente como primer parámetro al objeto mismo que estamos manejando, al que por convención llamamos `self`.

Y allí vemos que el cuerpo del método `__init__` lo que hace es crear los nombres `cateto1`

y `cateto2` *adentro del objeto*, apuntando a los objetos recibidos. Entonces, cuando instanciamos TriánguloRectángulo la primera vez, pasamos los valores 4 y 5 y en ese caso el `self` es el objeto que terminamos llamando t1 y guarda esos dos valores, y cuando la instanciamos por segunda vez, pasando los valores 10 y 1, `self` es el objeto que terminamos llamando t2, con estos dos otros valores en vez.

Usemos ahora el otro método que tenemos definido:

Cell 10

```
t1.hipotenusa()
```

```
6.4031242374328485
```

Cell 11

```
t2.hipotenusa()
```

```
10.04987562112089
```

Vemos que lo ejecutamos desde t1 y t2, y no pasamos ningún parámetro. Pero en la definición, arriba en la clase, recibe el parámetro `self`. Estamos en la misma situación que antes: como es el método de una clase, cuando lo ejecutamos desde una instancia de la clase Python insertará el objeto mismo como parámetro, que oportunamente usamos para la cuenta: cuando hacemos por ejemplo `self.cateto1 ** 2` estamos usando el nombre `cateto1` de adentro del objeto, que para t1 apuntará a 4 y para t2 apuntará a 10.

Al final de cuentas, lo que tenemos es un determinado código que se aplica a los valores que tiene cada instancia de esa clase. Exactamente el concepto con el que arrancamos arriba toda la explicación de clases, objetos, y POO.

Otro concepto muy útil cuando queremos modelar la realidad usando objetos es el de "herencia", generalmente utilizado cuando tenemos algunos comportamientos que son comunes a distintos tipos de objetos. Lo normal es encontrar una clase "padre" y muchas clases "hijas", pero Python soporta herencia múltiple, aunque es de uso más raro.

Cuando definimos una clase, entonces, si queremos que herede de otra incluiremos a esta última entre paréntesis en la definición de la primera. No vamos a entrar en detalle en este libro sobre cómo explotar todas las características del concepto de herencia, pero lo mencionamos para poder reconocer su uso cuando lo encontremos en algún código.

Particularmente, un caso que incluso ya mostramos en la Sección 1.4.4 es cuando creamos una excepción propia, que para que sea justamente una excepción utilizable por el lenguaje, la definimos heredando su comportamiento de alguna excepción integrada en el lenguaje:

Cell 12

```
class NoNegativos(ValueError):
    """Usada al encontrar números negativos, que no se puede."""
```

1.5.3. Módulos

Como vimos hasta ahora, las formas más comunes de encapsular código son las funciones y las clases. Entonces, pondremos código adentro de esas estructuras, que usaremos desde distintos puntos de nuestro programa. El paso natural siguiente es el de agrupar algunas de esas funciones y clases de nuestro programa en *módulos*, de manera de poder importar esos módulos de distintos lugares y tener acceso a las funciones y clases (y cualquier otra estructura) que pongamos allí.

Los módulos no son más que archivos de Python, sin tener casi ninguna otra restricción.

Mostremos un ejemplo para ver lo sencillo que es crear y usar un nuevo módulo de Python.

Pongamos el siguiente código en un archivo, al que llamaremos `perimetros.py`:

```
import math

dos_pi = 2 * math.pi

def cuadrado(lado):
    """Calcula el perímetro del cuadrado."""
    return 4 * lado

def círculo(radio):
    """Calcula el perímetro del círculo."""
    return dos_pi * radio
```

Vemos en el ejemplo que además de la definición de esas funciones tenemos otras líneas de código a "nivel de módulo", como el **`import`** o el cálculo para tener "dos π" a mano. Todo el código del módulo se ejecutará cuando lo importemos; se importará math, se definirá `dos_pi`, y también se definirán las dos funciones que usaremos luego.

Para usar ese módulo, como es el caso con cualquier otro módulo, sólo tenemos que importarlo. Entonces, en el mismo directorio que grabamos `perimetros.py`, abramos un intérprete interactivo y hagamos:

```
>>> import perimetros
>>> perimetros.círculo(12)
75.39822368615503
```

Necesitamos que el módulo esté en el mismo directorio donde abrimos el intérprete interactivo porque cuando hacemos el **`import`** Python va a buscar el nombre que indicamos en una serie de directorios, entre ellos el actual del proceso. Claro, podríamos poner nuestro módulo en algunos de los otros directorios donde Python busca, pero ello ya implicaría *instalar* nuestro módulo.

La otra opción para facilitar que podamos encontrar nuestro módulo es directamente agregar el directorio que necesitemos en la lista de lugares donde Python busca. Esto lo podemos hacer a través de la variable de entorno `PYTHONPATH` del sistema operativo, o incluso desde adentro de Python modificando `sys.path`.

Volviendo al ejemplo donde usamos nuestro módulo, vemos que lo importamos usando su nombre, y luego podemos acceder a su contenido usando el `.`, como ya vimos en otros casos. Esta no es la única manera de importar el módulo, también podemos utilizar otra forma en la que en vez de quedarnos con el nombre del módulo para trabajar, nos traemos directamente los nombres de las estructuras internas:

```
>>> from perimetros import círculo
>>> círculo(12)
75.39822368615503
```

Tengamos en cuenta que no cambia nada a la hora de importar el módulo en sí, no es más rápido, ni usa menos memoria, ni ejecuta menos código: la única diferencia es con qué nombres nos quedamos para trabajar.

Bien, ya sabemos agrupar nuestro código en módulos. El próximo paso es agrupar esos módulos en un próximo nivel.

La estructura para agrupar módulos se llama *paquete*, que no son más que directorios.

Para probar esto, creemos un directorio `geom` y pongamos nuestro archivo `perimetros.py` allí. Eso es todo; ahora para importar ese módulo tenemos que especificar el paquete. Veamos las distintas formas de terminar ejecutando la función `círculo` en esta nueva situación: indicando el paquete y el módulo, trayendo el módulo del paquete, y trayendo directamente la función (de nuevo, en los tres casos el módulo se importa exactamente igual, sólo cambia con qué nos quedamos para trabajar).

```
>>> import geom.perimetros
>>> geom.perimetros.círculo(12)
75.39822368615503
>>> from geom import perimetros
>>> perimetros.círculo(12)
75.39822368615503
>>> from geom.perimetros import círculo
>>> círculo(12)
75.39822368615503
```

Si en un directorio/paquete ponemos un archivo con el nombre especial `__init__.py`, este archivo se ejecutará cuando importemos el paquete o cualquier módulo de ese paquete, lo cual es muy práctico para cualquier tipo de inicialización que necesitemos realizar.

1.6. Cómo pedir ayuda

Una de las grandes cosas buenas de Python es su Comunidad, alrededor del mundo y en infinidad de idiomas.

Siempre vamos a encontrar un foro, una lista de correo, o algún recurso gratuito puesto a disposición por gente tratando de ayudar. Y la misma comunidad se autorregula para seguir siendo sana, tanto informalmente como formalmente, por ejemplo creando la Python Software Foundation en Estados Unidos, o la Asociación Civil Python Argentina [18], organizaciones que aprueban y alientan la participación de todes. Nuestra comunidad esta basada en respeto mutuo, tolerancia y fomento, y estamos trabajando para ayudar a cada uno y a cada una a vivir a la altura de estos principios. Queremos que la comunidad sea más diversa [19]: quien quieras que seas, y cualquiera sean tus orígenes, te recibiremos.

Los canales de comunicación virtuales son el principal medio de comunicación e integración de la Comunidad, y es un buen punto de entrada para cuando tenemos consultas y preguntas. Aprender a utilizar estos mecanismos es un activo tan importante en un programador o una desarrolladora de software como saber tal función, tal estructura del lenguaje, o tal detalle de implementación.

Es por esto que queremos hacer mucho énfasis en que busquen como interactuar con la Comunidad, cuales grupos de usuaries tienen cerca, qué lista de correo o foro les apetece más, y empiecen a participar allí. Si ya tienen consultas pueden hacerlas, pero también es muy valioso leer las preguntas de otres y las respuestas que se ofrecen, e incluso tratar de responderlas nosotres. Esta es una de las mejores formas de profundizar nuestro conocimiento en Python.

Te recomendamos explorar el sitio de Python Argentina [20], o entrar directamente en su grupo de Telegram [21] o en el foro [22]. Pero no dejes de de revisar si hay alguna comunidad local más cerca de donde estés [23].

Y siempre está la comunidad global de Python, con muchísimos recursos en su sitio [24], en particular las listas de correo [25] y el foro [26], pero claro, esto es todo en inglés.

Parte II
Temas específicos

En esta Parte desarrollaremos capítulos que abordan temas específicos de aplicación de herramientas de Python. Cada capítulo está autocontenido, de forma que el lector o lectora puede acceder directamente al tema de interés sin realizar un recorrido secuencial de los capítulos, aunque serán utilizados conceptos y herramientas tratados en la Parte I.

Parte III
Apéndices

A | Zen de Python

Incluimos aquí las frases traducidas correspondientes al Zen de Python [4].

- Bello es mejor que feo.
- Explícito es mejor que implícito.
- Simple es mejor que complejo.
- Complejo es mejor que complicado.
- Plano es mejor que anidado.
- Espaciado es mejor que denso.
- La legibilidad es importante.
- Los casos especiales no son lo suficientemente especiales como para romper las reglas.
- Sin embargo la practicidad le gana a la pureza.
- Los errores nunca deberían pasar silenciosamente.
- A menos que se silencien explícitamente.
- Frente a la ambigüedad, evitar la tentación de adivinar.
- Debería haber una, y preferiblemente solo una, manera obvia de hacerlo.
- A pesar de que esa manera no sea obvia a menos que seas Holandés.
- Ahora es mejor que nunca.
- A pesar de que nunca es muchas veces mejor que *ahora* mismo.
- Si la implementación es difícil de explicar, es una mala idea.
- Si la implementación es fácil de explicar, puede que sea una buena idea.
- Los espacios de nombres son una gran idea, ¡tengamos más de esos!

Bibliografía

[1] URL: https://creativecommons.org/licenses/by-nc-sa/4.0/deed.es.

[2] python-dev <python-dev at python.org>. *Index of Python Enhancement Proposals (PEPs)*. 13 de jul. de 2000. URL: https://www.python.org/dev/peps/.

[3] URL: https://www.python.org/psf/.

[4] Tim Peters. *The Zen of Python*. 19 de ago. de 2004. URL: https://www.python.org/dev/peps/pep-0020/.

[5] URL: https://docs.python.org/dev/reference/index.html.

[6] URL: https://docs.python.org/dev/library/index.html.

[7] URL: https://www.python.org/downloads/.

[8] URL: http://vim.fisadev.com/.

[9] URL: https://wiki.python.org.ar/ides/.

[10] URL: https://ipython.org/.

[11] URL: https://docs.python.org/dev/library/stdtypes.html#string-methods.

[12] URL: https://docs.python.org/dev/library/string.html#format-string-syntax.

[13] URL: https://www.youtube.com/watch?v=jAZ-NyAwpsg.

[14] URL: https://www.python.org/dev/peps/pep-0622/.

[15] URL: https://docs.python.org/dev/library/exceptions.html#exception-hierarchy.

[16] URL: https://docs.python.org/dev/library/exceptions.html#concrete-exceptions.

[17] URL: https://docs.python.org/dev/reference/datamodel.html#special-method-names.

[18] URL: https://ac.python.org.ar/.

[19] URL: https://ac.python.org.ar/diversidad/index.html.

[20] URL: https://www.python.org.ar/.

[21] URL: https://t.me/pythonargentina.

[22] URL: https://charlas.python.org.ar/.

[23] URL: https://wiki.python.org/moin/LocalUserGroups.

[24] URL: https://www.python.org/.

[25] URL: https://www.python.org/community/lists/.

[26] URL: https://python-forum.io/.