USERS

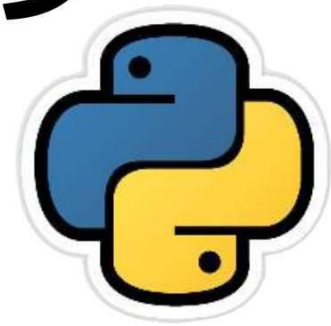
VOL. I

Programación en python

ENTORNO DE PROGRAMACIÓN - SINTAXIS
ESTRUCTURAS DE CONTROL

VOL. I

Programación en python



ENTORNO DE PROGRAMACIÓN - SINTAXIS
ESTRUCTURAS DE CONTROL

Título: Programación en Python - Vol. I / **Autor**: Celeste Guagliano
Coordinador editorial: Miguel Lederkremer / **Edición**: Claudio Peña
Diseño y Maquetado: Marina Mozzetti / **Colección**: USERS ebooks - LPCU290

Guagliano, Celeste
Programacion en Python 1 : entorno de programación : sintaxis, estructuras de control / Celeste Guagliano. - 1a ed . - Ciudad Autónoma de Buenos Aires : Six Ediciones, 2019.
Libro digital, PDF - (Programacion en Python ; 1)

Archivo Digital: descarga y online
ISBN 978-987-4958-09-9

1. Lenguajes de Programación. I. Título.
CDD 005.2

CURSO DE PROGRAMACION PYTHON

PRÓLOGO

Python es un lenguaje sumamente versátil y robusto que fue concebido como la conjunción de las mejores características de otros lenguajes.

Sin dudas es un lenguaje de mucho crecimiento, el cual se ve fomentado por la alta demanda de programadores especializados en este lenguaje, ya sea desde grandes empresas multinacionales como desde el ambiente científico.

La idea detrás de estos ebooks, es permitir un acercamiento amigable a este lenguaje de programación no solo como una herramienta teórica sino también desde los planteos de ejemplos y ejercicios que te permitirán afianzarte en la programación desde la misma práctica.

Todos los que hemos estudiado alguna vez cualquier lenguaje de programación en forma autodidacta, sabemos que ningún texto será capaz de entregarnos la totalidad de las herramientas que necesitaremos para enfrentarnos al desafío de escribir nuestros propios códigos. Sin embargo, este texto pretende entregarnos las bases y permitirnos explorar un sinfín de ejemplos, buenas prácticas y errores comunes para facilitarnos la tarea a la hora de sentarnos frente al teclado y comenzar a codificar nuestros programas.

Particularmente escribir esta serie de ebooks fue un gran desafío personal que disfruté a lo largo de todo el trayecto, intentando pensar que datos me hubieran resultado de utilidad en el momento en el que me encontraba dando mis primeros pasos en Python.

Celeste Guagliano

Acerca de la autora

Celeste Guagliano es Ingeniera en Automatización y Control Industrial, docente universitaria y entusiasta de la programación y nuevas tecnologías. Se desempeña hace 5 años como docente universitaria en el ámbito de la programación, en donde enseña diferentes paradigmas y lenguajes tales como Assembler, C, Java y Python. Su interés por la programación comenzó a la temprana edad de 9 años, en 1992 cuando tuvo acceso por primera vez a una computadora y pudo dar sus primeros pasos en GWBasic. En el primer volumen de esta serie se contó con la colaboración de Cecilia Jarne, doctora en Física de la Facultad de Ciencias Exactas, Universidad Nacional de La Plata.

Sobre este curso

Python es un lenguaje de programación multiplataforma, consistente y maduro, utilizado por numerosas empresas internacionales. Se utiliza en múltiples campos tales como aplicaciones web, juegos y multimedia, interfaces gráficas, networking, aplicaciones científicas, inteligencia artificial y muchos otros.
En esta serie de ebooks sobre programación en Python el lector encontrará todo lo necesario para iniciarse o profundizar sus conocimientos en este lenguaje de programación.
El curso se compone de tres volúmenes, orientados tanto a quien recién se inicia en este lenguaje, como a quien ya está involucrado y quiere profundizar sus conocimientos de Python.

Volumen I

Se realiza una revisión de las características de este lenguaje, también se entregan las indicaciones para instalar el entorno de desarrollo y, posteriormente, se analizan los elementos básicos de la sintaxis y el uso básico de las estructuras de control, finalizando con una serie de códigos de ejemplo explicados en detalle.

Volumen II

Se presenta el paradigma de programación orientada a objetos con todas sus implicancias: clases, herencia y todo el campo de posibilidades que nos abre comenzar a utilizar este paradigma en Python.

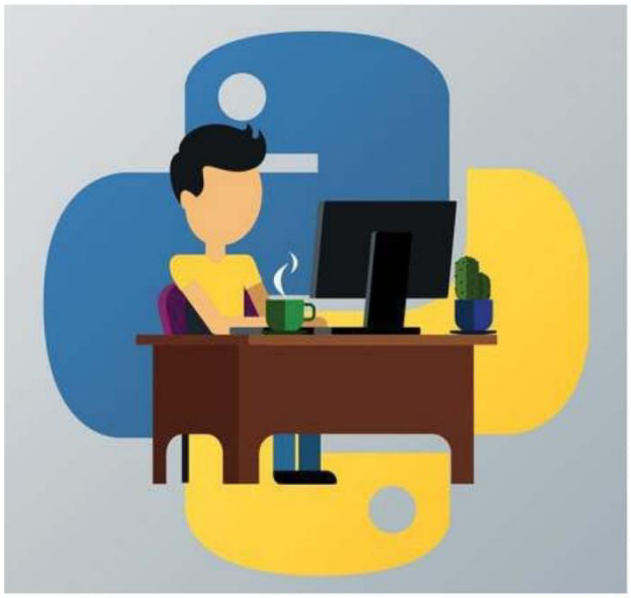
Volumen III

Orientado a la aplicación de Python en proyectos, veremos ejemplos de aplicación en Raspberry pi y micropython entre otros.
En estos tres volúmenes iremos aumentando gradualmente la complejidad de los temas para que el recorrido de aprendizaje resulte ameno y motivar.

¡Exitos en este nuevo desafio!

Sumario

Vol. I



Introducción

¿QUÉ ES PYTHON?

Python es un lenguaje de programación, y ¿qué es un lenguaje de programación? Todos nosotros tenemos una lengua nativa y probablemente sepamos alguna lengua más, tal podría ser el caso de español como lengua nativa e inglés como segunda lengua; queramos o no, si manejamos computadoras y sabemos algo de programación, entonces sabemos algo de inglés. Un lenguaje de programación es un idioma que la computadora conoce y nos sirve para ordenarle distintas acciones.

Pero Python no solo es un lenguaje de programación, sino que además es *interpretado*; ¿qué significa esto?, que la computadora por sí sola no conoce el lenguaje y lo comprende, sino que necesita un *intérprete*. Esto sería similar a visitar un país del cual no conocemos ni una palabra de la lengua nativa, pero viajamos con una persona que conoce nuestra lengua y la de ese país, y nos hace de traductor para que logremos entendernos con las demás personas.

Un interrogante que se presenta en este punto es: ¿hay otros tipos de lenguajes que no sean interpretados? La respuesta es sí. El mundo de la programación se divide en dos clases de lenguajes: interpretados y compilados. ¿En qué reside la diferencia? En un lenguaje compilado, el programador escribe el código en el lenguaje de su preferencia, siempre y cuando sea un lenguaje que se compile, y luego, por medio de un compilador, ese código

Python es un lenguaje de programación interpretado que posee muchas virtudes y ventajas de uso así como también numerosas aplicaciones prácticas. En este primer capítulo conoceremos todo lo necesario para comenzar a trabajar con Python.

01

se "traduce" al lenguaje ensamblador que entiende la computadora. La ventaja de esto es que el código ya se encuentra totalmente traducido y su ejecución es veloz. En cambio, con un lenguaje interpretado, el intérprete traduce el código para que la computadora lo comprenda a medida que se va ejecutando, y esto puede llegar a resultar un poco más lento en la ejecución que un código compilado.

Pero entonces, ¿por qué elegir un lenguaje interpretado? Una de las principales ventajas es que se puede ir probando el código a medida que lo vamos escribiendo, un intérprete no sabe ni le importa cuándo termina el código para hacer su trabajo. En cambio, un compilador realiza su tarea hasta que encuentra la instrucción de *fin*, por lo que si el código que queremos compilar no está completo, no lograremos compilar y ejecutar nuestro programa. Esta diferencia hace que programar en un lenguaje interpretado sea mucho más dinámico, y de esta forma se optimiza el tiempo de programación y depuración del código escrito.

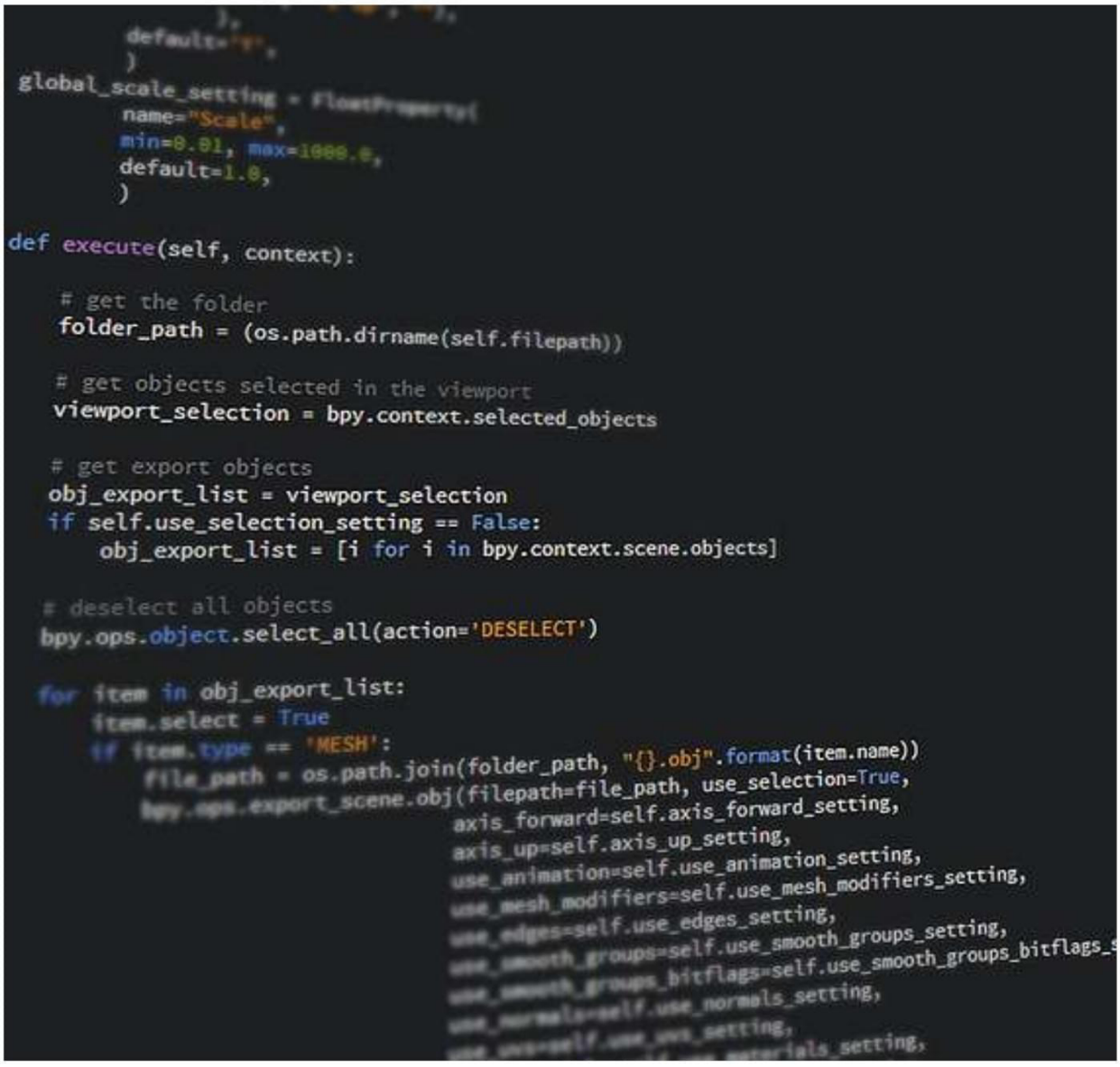
GUIDO van ROSSUM y la particularidad del nombre PYTHON

Guido van Rossum, informático holandés, en sus vacaciones de Navidad de 1989 decidió poner en marcha un proyecto personal que consistió en el desarrollo de un lenguaje de programación interpretado.
Guido decidió utilizar como base para su lenguaje todas las características que le gustaban de otros lenguajes con los que había trabajado y, con esto en mente, comenzó a escribir en C un intérprete para su futuro lenguaje de programación.
Guido decidió bautizar este lenguaje con el nombre Python en honor a su serie favorita de televisión *Monty Python's Flying Circus*.

¿POR QUÉ NOS CONVIENE APRENDER PYTHON?

Las ventajas que presenta el uso de Python son las siguientes: facilidad de uso, legibilidad de código, integración con sistemas embebidos, optimización del lenguaje para trabajar con múltiples núcleos en tareas paralelas, variedad de bibliotecas y una amplia comunidad de usuarios consolidada a nivel mundial, entre muchas otras.

Veamos en detalle estos puntos a favor de Python.



Lenguaje sencillo de aprender

Vamos a ver que Python presenta una sencillez tal que cualquier persona que realiza una pequeña inversión de tiempo puede comenzar a crear programas sencillos en este lenguaje. Esto se debe, entre muchas cosas, a la gestión automática de memoria o las operaciones sencillas de lectura y escritura, en las que se diferencia de otros lenguajes. Tal es el caso de C, en el cual tanto la asignación de memoria como otras características son mucho más engorrosas de programar.

Todos los que nos hayamos enfrentado con el aprendizaje de un lenguaje nuevo de programación sabemos que, según como nos enfoquemos en esta tarea, puede resultar muy emocionante o terriblemente tediosa. Más aún si queremos aprender por nuestra cuenta.

Sin embargo, Python fue desarrollado pensando en que su aprendizaje resultara sencillo, incluso como primer lenguaje de programación. ¿Por qué? Porque su sintaxis es muy sencilla. Como se utilizan expresiones comunes, se escribe menos código y se obtienen resultados más rápido. O sea, Python requiere menos líneas de código para realizar tareas básicas que si programáramos las mismas tareas en Java o C++. Otro plus que nos ofrece Python tiene que ver con su librería estándar, la cual permite ejecutar otras funciones y tareas más complejas con mayor facilidad que otros lenguajes.

Por todo lo mencionado, vemos que Python es un lenguaje ideal si queremos encarar la tarea de aprendizaje por nuestra cuenta.

Sirve como base teórica de programación

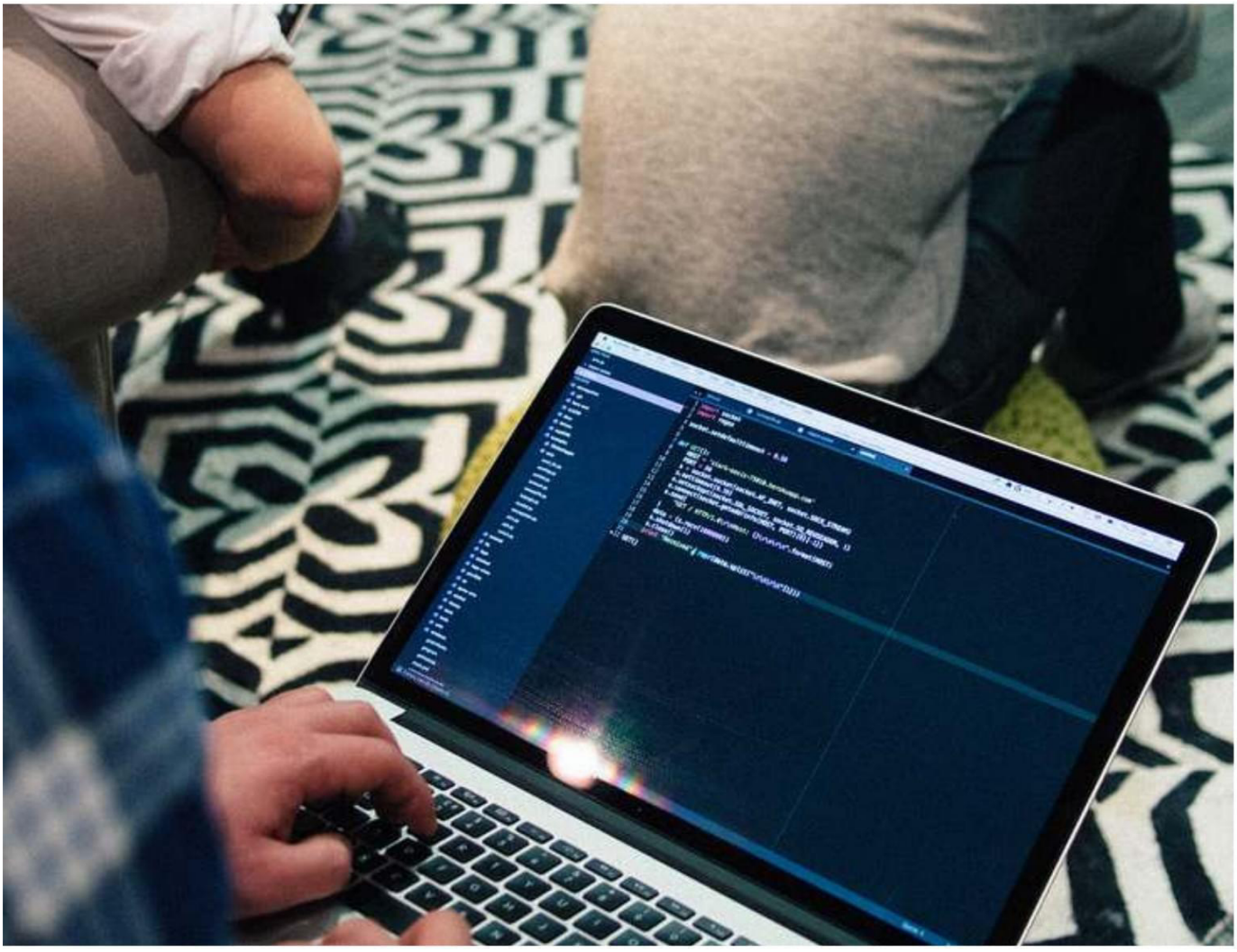
Python es un lenguaje orientado a objetos, pero cuya versatilidad nos permite utilizarlo aplicando diferentes paradigmas de programación. Lo interesante de Python es que su sencillez nos permite tanto aprender a programar si recién nos iniciamos, como también aprender las bases de un paradigma de mayor complejidad, como es la programación orientada a objetos. De esta forma, si luego queremos migrar a otro lenguaje, como **Java**, **C++**, **Ruby** u otros, el paso nos resultará más sencillo que si quisiéramos aprender directamente el paradigma en estos lenguajes cuya sintaxis es más compleja. Es decir, Python puede ser el primer paso como programador ofreciéndonos una base sólida que nos ayudará en pasos siguientes.

La demanda del lenguaje es alta

Si queremos dedicarnos a la programación, es fundamental aprender Python, ya que grandes compañías multinacionales, como **Google**, **Nokia** e **IBM**, lo utilizan. Es decir, resulta sencillo entrar al mercado laboral sabiendo programar en Python.

Se emplea en desarrollo web

Python se emplea en el desarrollo de aplicaciones y sitios web. En la actualidad, existen diversos frameworks para que el proceso de desarrollo sea más sencillo. Además de herramientas para el desarrollo web, Python puede usarse en el desarrollo de juegos y se utiliza ampliamente en el mundo científico, incluso la NASA aplica este lenguaje de programación.



Integración con sistemas embebidos

Algunas plataformas, como **Raspberry Pi**, se basan en Python. También cabe mencionar al proyecto **MicroPython**. Esto representa una alternativa a plataformas tales como Arduino, con un gran potencial para proyectos de diferente índole, que facilita sustancialmente la programación y el testeo de sistemas con hardware integrado.

Facilidad de escritura de código para diferentes hilos

Los lenguajes diseñados antes de que se masificaran las plataformas multiprocesador o multinúcleo son muy complejos de programar para poder aprovechar el potencial de paralelizar tareas; en Python en cambio, el código asincrónico se gestiona de manera sencilla.

Muchas bibliotecas disponibles

Hay una gran variedad de bibliotecas disponibles en el mundo Python, desde manejo matemático hasta procesamiento de imagen y muchas funcionalidades más.

Tiene una gran comunidad que lo respalda

Una de las ventajas de aprender un lenguaje consolidado y de mucha popularidad como Python es la gran comunidad de usuarios que se consolidó a su alrededor. Recordemos que Python es una herramienta Open Source, lo que significa que a mayor comunidad, mayor desarrollo tendrá el lenguaje.

Una gran comunidad de usuarios nos garantiza mucha información disponible, número de bibliotecas creciente, proyectos y código disponible si buscamos por internet, además de foros activos en donde podemos consultar y evacuar dudas. Esto hace que tanto el aprendizaje de proyectos como su ejecución sean muy dinámicos.

APLICACIONES DE PYTHON

En este apartado veremos un poco de historia de diferentes aplicaciones y desarrollos en los que se utilizó y se sigue usando Python, como muestra del potencial de este lenguaje en el mercado a lo largo del tiempo.

Inteligencia artificial (AI)

Por todas las características que mencionamos con anterioridad y además por tratarse de un lenguaje de código abierto, Python es un aliado perfecto para la inteligencia artificial.

Permite plasmar ideas complejas con unas pocas líneas de código, lo que no es posible con otros lenguajes.

Algunas bibliotecas disponibles en Python que podemos mencionar son **Keras** y **TensorFlow**, que contienen mucha información sobre las funcionalidades del aprendizaje automático.

Además, existen bibliotecas proporcionadas por Python, que se usan mucho en los algoritmos de inteligencia artificial, como **Scikit**, una biblioteca gratuita de aprendizaje automático que presenta varios algoritmos de regresión, clasificación y agrupamiento.

Big Data

Python resulta muy útil, y su uso está muy extendido en el análisis de datos y la extracción de información útil para empresas .mediante Big Data

Además de su simplicidad, que es una gran ventaja, Python cuenta con bibliotecas de procesamiento de datos como **Pydoop**, que son de gran ayuda para los profesionales, ya que se puede escribir un código de **MapReduce** en Python y procesar los datos en el clúster **HDFS**.

Otras bibliotecas, como **Dask** y **PySpark**, simplifican aún más el análisis y la gestión de datos. Python es rápido y fácilmente escalable, características fundamentales al querer procesar un gran flujo de datos y, de esta forma, resulta útil para generar información en entornos de tiempo real y convertir esa información a los lenguajes usados en **Big Data**.

Data Science

Al contar con paquetes numéricos, como **Pandas** y **NumPy**, es natural que los investigadores hayan comenzado a trabajar con Python dejando de hacerlo con software de simulación y procesamiento de datos pagos como **MATLAB**.

Python se ocupa de los datos tabulares, matriciales y estadísticos, e incluso los visualiza con bibliotecas populares como **Matplotlib** y **Seaborn**.

Frameworks de pruebas

El *testing* es otra de las actividades en las que Python llegó para quedarse. Python es ideal para validar ideas o productos, debido a sus numerosos frameworks integrados que ayudan a depurar el código, y ofrecen flujos de trabajo y ejecución rápidos. Herramientas de testing, como **Unittest**, **Pytest** y **Nose test**, facilitan las pruebas. Python, además, admite pruebas entre plataformas y navegadores con diferentes marcos, como PyTest y Robot.

El testing, una de las tareas más arduas que nos pueden encomendar, se simplifica considerablemente con el uso de Python.

Desarrollo web

Como ya dijimos en apartados anteriores, Python permite construir mucho más con menos líneas de código, por lo que se crean prototipos de forma más eficiente.

El *framework* **Django**, proporcionado por Python, presenta la ventaja de poder utilizarse para crear aplicaciones web dinámicas y muy seguras.

El lenguaje Python también se usa para hacer *scraping*, o sea, obtener información de otros sitios web. Algunas aplicaciones construidas con este tipo de frameworks son Instagram, **Bit Bucket**, **Pinterest**.

Instagram

Con todas las ventajas que venimos nombrando de Python, no parece extraño que un sitio de la magnitud y con el volumen de datos que maneja Instagram utilice Python.

Al ser un lenguaje en el cual es sencillo realizar un desarrollo, es muy simple de gestionar y puede manejar sin problemas el gran volumen de visitas que recibe diariamente, Python se vuelve ideal para los ingenieros que trabajan en estas tareas. Todas las ventajas que hemos nombrado, permiten al equipo de trabajo enfocarse en la experiencia de usuario.

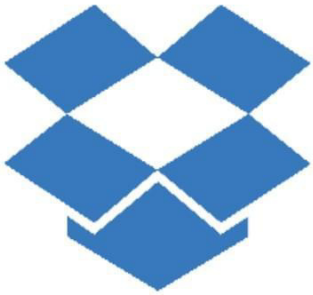
Si bien todo el código que hace funcionar a Instagram no está escrito en Python, y se utilizan también otros lenguajes tanto en el *frontend* como en el *backend*, podemos decir que el corazón de Instagram se encuentra escrito en Python.

Pinterest

Pinterest, uno de los sitios de imágenes y proyectos más difundidos de la actualidad, también utiliza Python.

El motor de la plataforma web de Pinterest se encuentra desarrollado completamente con el lenguaje de programación Python. Además, el framework Django se utiliza en la capa de aplicación o backend del sitio.

Dropbox



La mayor parte del código de **Dropbox** está escrito en Python tanto para el cliente Desktop como para la aplicación. Además se utiliza en los controladores de la web, garantizando que Dropbox funcione perfectamente en cualquier sistema operativo.

Battlefield 2

Ya hablamos antes de la versatilidad de Python, por lo que es natural pensar que no solo se utiliza para desarrollos web. *Battlefield* es un juego de batalla en primera persona, donde el jugador toma control de distintos equipos bélicos y vehículos para realizar misiones. Es un juego con buena calidad gráfica y de historia interesante que además está desarrollado completamente en Python, desde el motor de juego hasta la mayoría de las animaciones.

Google App Engine

Google App Engine es un servicio de desarrollo web que permite la creación de aplicaciones web y móviles, conectándose con servicios en la nube, tales como Google Cloud, y obteniendo resultados de eficiencia aceptables y profesionales.

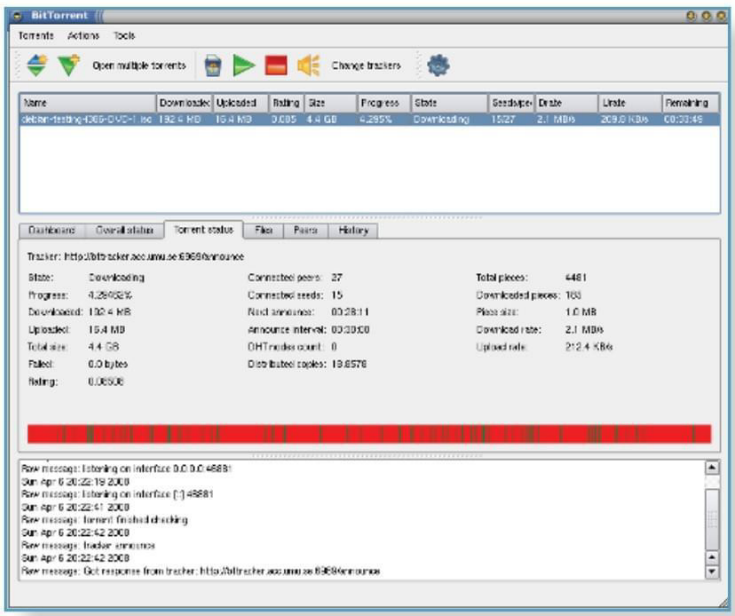
Parte de su código está desarrollado en Python y además acepta el uso de Python para el desarrollo web, de esta forma permite el uso de frameworks para la construcción de sitios escalables y de un volumen de tráfico considerable.

Ubuntu Software Center

El centro de software de **Ubuntu** es un sistema de paquetes gráficos del sistema operativo Ubuntu, el cual se encarga de administrar, instalar, reemplazar o eliminar aplicaciones. Se encuentra desarrollado íntegramente en Python, lo que le da estabilidad y crea una experiencia de usuario muy buena.

BitTorrent

BitTorrent es un conocido protocolo de intercambio de datos que permite guardar la información que se está descargando sin temor a perderla, pausar la descarga y reanudarla mas tarde, incluso luego de varios días y de apagar y volver a encender el equipo, sin pérdidas de datos.



Este protocolo y su aplicación fueron completamente creados y diseñados en Python, se lanzó en 2001 y, si bien con los años y las diferentes versiones se realizaron algunas reestructuraciones, su base se mantiene y es un claro ejemplo de lo eficiente y poderoso que puede resultar Python en el desarrollo de grandes proyectos.

Panda 3D

Panda 3D es un motor para juegos que contiene gráficos y sonido. Está especialmente orientado para la creación de juegos en 3D. Su popularidad es muy alta debido a que estamos hablando de software libre por lo que cualquiera de nosotros podría utilizarlo sin tener que pagar el derecho de uso. No obstante, grandes corporaciones, como Disney, también hacen uso de su código base.

Panda 3D se encuentra desarrollado en Python y C++, pero Python es quien aporta sus ventajas, y es reconocido como un motor eficiente que permite la creación de juegos de alta calidad.

La NASA

La Administración Nacional de Aeronáutica y del Espacio utiliza Python para el desarrollo de aplicaciones que se ejecutan en el desarrollo de sus proyectos. Algunas de las aplicaciones que podemos mencionar son: un repositorio para almacenamiento de datos de CAD (diseño asistido por computadora) que se usa en los viajes espaciales; un sistema de gestión, integración y transformación, que la NASA considera que se perfila como la base fundamental para la asistencia en el ámbito de la ingeniería en los próximos años; y por último, podemos mencionar **OpenMDAO**, una herramienta para resolver problemas de optimización de diseños multidisciplinarios.

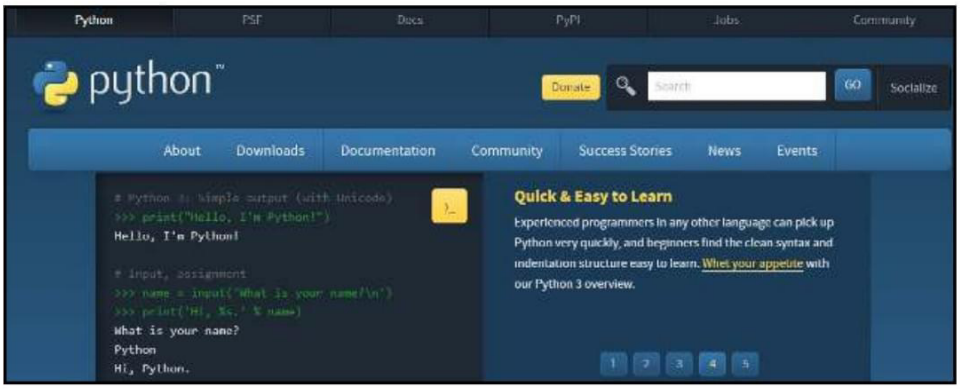
Instalación

INSTALAR EN WINDOWS

A continuación se explicará cómo realizar la instalación de Python desde cero en Windows: desde elegir la versión adecuada hasta comprobar que la instalación se realizó correctamente, pasando por todas las opciones necesarias para que el intérprete funcione sin complicaciones.

1

En primer lugar, ingrese a la página www.python.org.

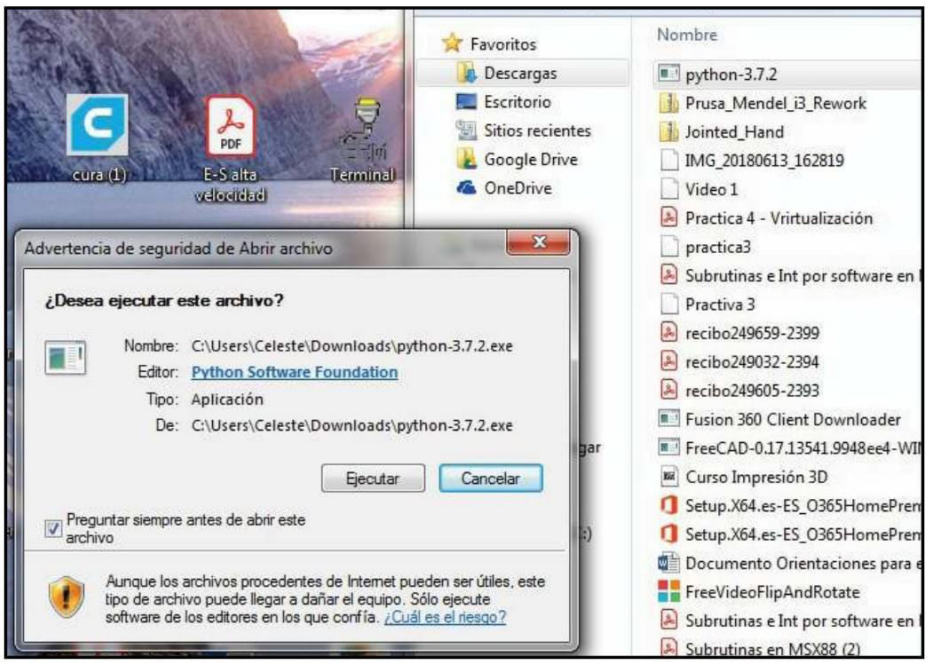


2

Desde la página podrá acceder a la descarga del instalador de la versión que haya decidido utilizar. Para ello haga clic sobre **download Download Python 3.x.x**.

Si nuestro sistema de trabajo es alguna distribución de Linux o MAC, probablemente ya tengamos preinstalada alguna versión de Python; no obstante, veremos cómo actualizar a la versión deseada. Si en cambio trabajamos en Windows, veremos cómo realizar la instalación desde cero.

02



3

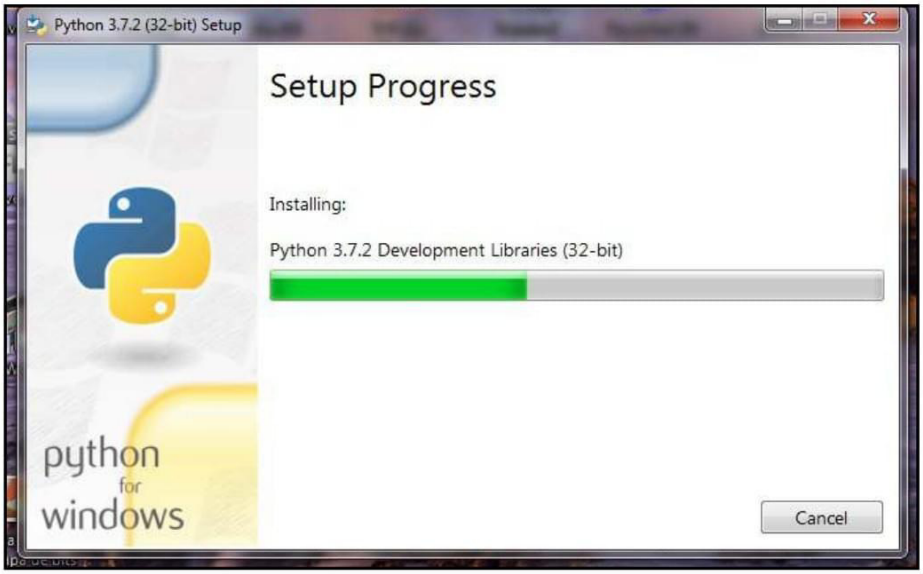
Una vez realizada la descarga, ejecute el instalador.

4

Al comenzar el proceso de instalación, tilde la opción **Add Python 3.x.x to PATH**, y luego haga clic en el botón **InstallNow**.

5

Verá el avance de la instalación en todo momento, esto demora varios minutos en Windows.



6

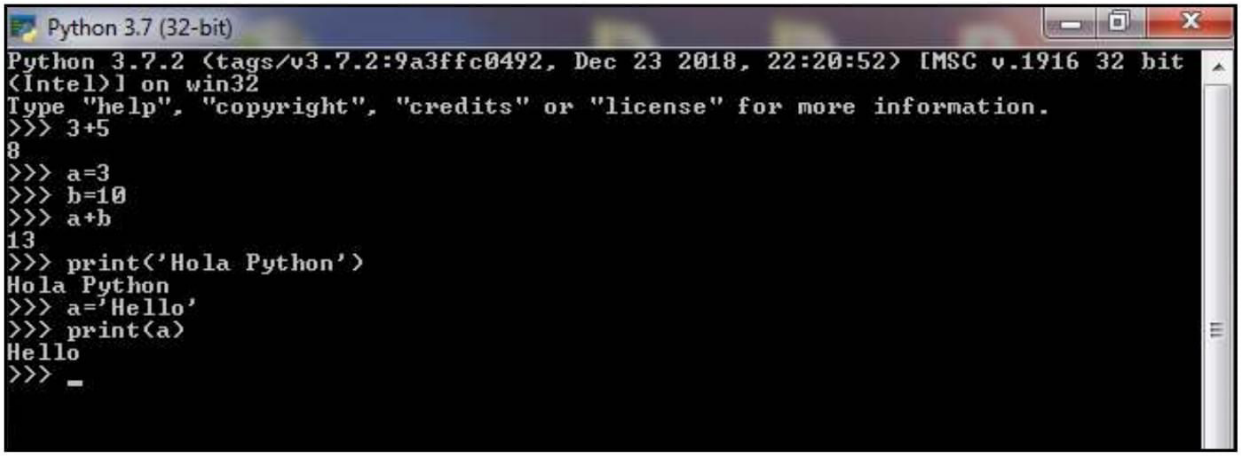
Al finalizar el proceso de instalación, aparecerá un cartel que informa que se ha realizado con éxito, y ya estará en condiciones de utilizar Python en su sistema Windows.

¿Qué ocurre si no agregamos PYTHON al PATH de Windows?

Lo que ocurrirá es que al querer ejecutar Python nos aparecerá un error del tipo: *Python no se reconoce como un comando interno o externo*. **Para solucionar este inconveniente, si nos olvidamos de tildar la opción en el momento de la instalación y no agregamos Python al PATH de Windows, tendremos que realizarlo en forma manual.**

TESTEO DE PYTHON EN WINDOWS

Para comenzar a utilizar Python, ingresamos desde el acceso directo en el **menú inicio** o bien desde el símbolo del sistema tecleando **Python3**. La pantalla del intérprete se verá muy similar a la terminal de Linux o a la vieja pantalla de DOS.



Una vez en esta pantalla, podemos comenzar a experimentar con códigos sencillos.

Algunos ejemplos sencillos para ver el correcto funcionamiento de Python se basan en realizar algunos cálculos, declarar variables y mostrar datos por pantalla. Lo que se visualiza con los símbolos **>>>** previos al texto es lo que se ingresa por teclado, y lo que se simboliza en un renglón sin estar acompañado de dichos símbolos es el resultado obtenido luego de presionar **ENTER**:

```
>>>5+3
8
>>>print('Hola Python')
Hola Python
>>>a=3
>>>b=12
>>>a+b
15
>>>a='Hello'
>>>print(a)
Hello
```

¿QUÉ VERSIÓN INSTALAR?

Si nunca utilizamos Python y tenemos que decidir qué versión instalar, la opción más lógica es elegir la versión 3.

Pero ¿qué sucede si ya veníamos trabajando con Python 2 y queremos actualizar a la versión 3?

El pasaje no es trivial, ya que Python 3 tiene muchas diferencias sustanciales con Python 2, y es casi seguro que nuestro código no funcionará.

Por eso a la hora de decidir qué versión utilizar, optaremos por la versión 2 o la 3 dependiendo de las características del proyecto que encaremos. Si vamos a continuar un proyecto que ya había sido comenzado en Python 2, entonces optaremos por esa versión, en caso contrario optaremos por la versión 3.

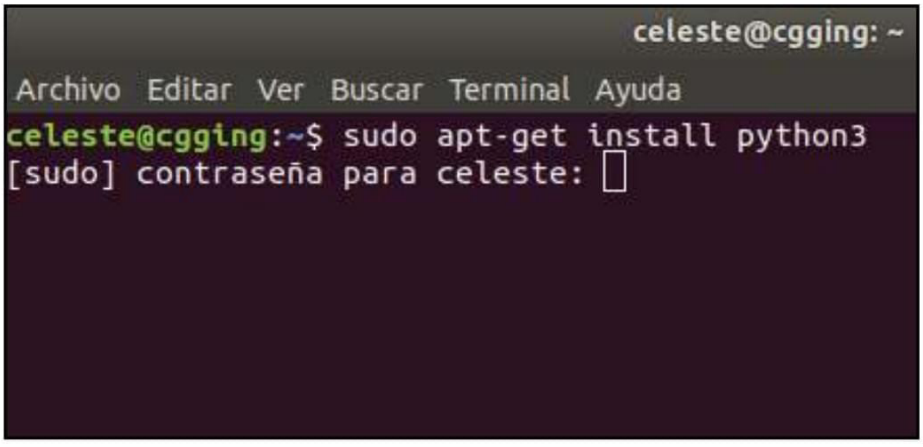
INSTALAR PYTHON EN LINUX

Como se dijo con anterioridad, Python viene preinstalado tanto en Linux como en MAC OS. No obstante, dependiendo de la versión de Python que queramos utilizar y de la versión de Linux o MAC con la que contamos, se deberá realizar la actualización o no de la versión instalada.

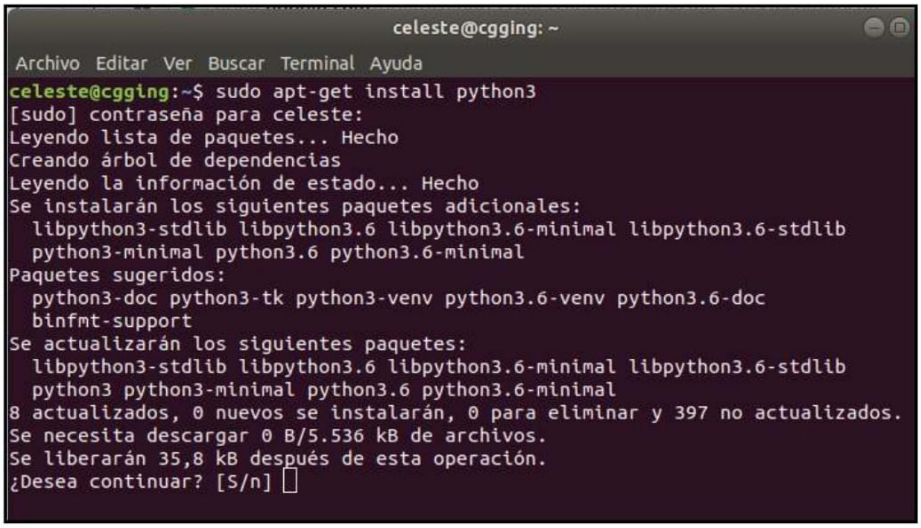
En este caso, utilizamos como sistema base **Ubuntu 18.04**.

1 Revise la versión de Python que viene instalada con su sistema operativo. Para ello, debe tipear desde la terminal de Linux: **python3 --version**. Si Python 3 se encuentra ya instalado en el sistema, se verá la información de la versión, en caso contrario el sistema informará que no halló dicho paquete.

2

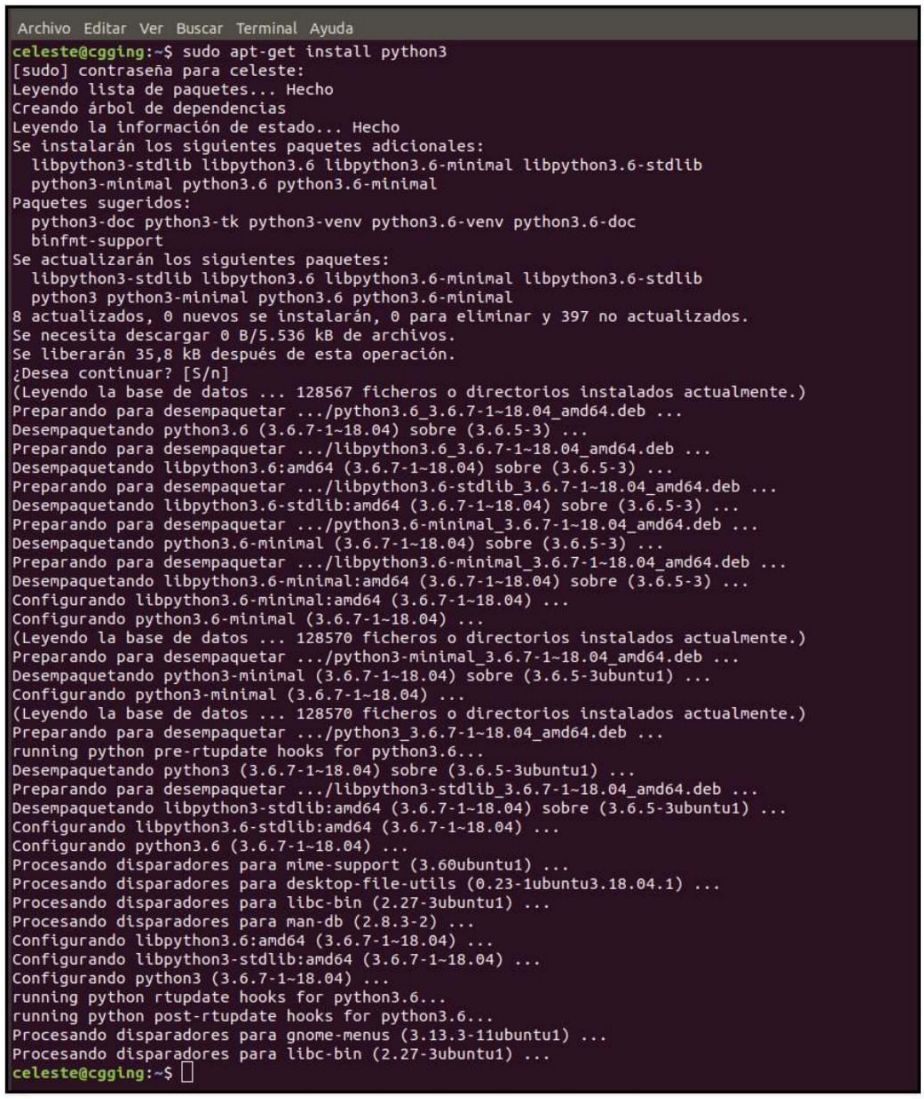


Para instalar Python 3 en Linux, desde la terminal escriba: **sudo apt-getinstall python3**. Le pedirá la clave del usuario para proceder con la instalación.



3

Linux informará sobre el paquete encontrado y deberá confirmar la instalación con **S** y **ENTER** o simplemente **ENTER** para ratificar.



4

El proceso de instalación en Linux dura algunos segundos y se muestra todo el proceso por la terminal.

5

Al finalizar el proceso se vuelve a ver el cursor en la terminal de Linux.

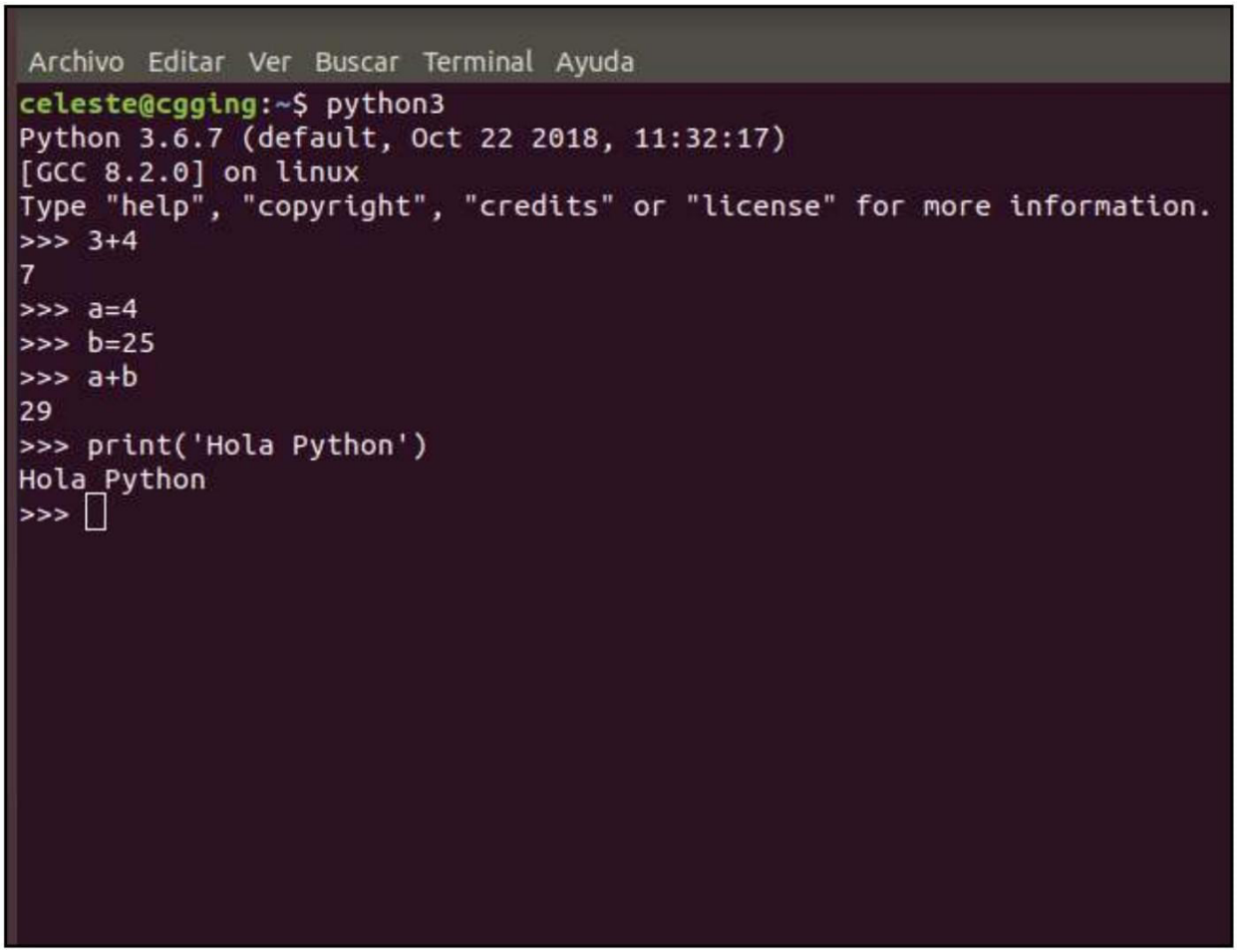
TESTEO DE PYTHON EN LINUX

Ahora que ya se completó la instalación de Python 3 en el sistema, se pueden realizar algunos testeos.

Para comenzar a utilizar Python, ingresamos a la terminal de Linux y escribimos:

```
Python3
```

y de esta manera iniciaremos el intérprete de Python. De la misma forma que en Windows, se pueden realizar algunos testeos básicos, tales como algunos cálculos, declaración de variables, mostrar texto por pantalla:



Aquí vemos el testeo básico de Python en Linux.

INSTALAR PYTHON EN MAC OSX

En Mac al igual que en Linux, Python viene preinstalado, por lo que de la misma forma que en Linux, se deberá comprobar la versión y proceder a la instalación de la que deseamos en el caso que no coincida con la que ya se encuentra en el sistema.

Para realizar la instalación, los pasos son muy similares a la instalación que vimos en Linux, ya que el entorno de trabajo es muy similar.

1 Ingrese a la terminal de Mac. Para ello, escriba **Terminal** en Finder o Spotlight.

2 Compruebe la versión de Python escribiendo en la terminal: **python3 --version**. Si no se encuentra la versión 3 instalada, proceda a la instalación en el siguiente paso.

3 Ingrese a la página www.python.org y realice la descarga de la versión de Python deseada.

4 Realice la instalación y compruebe nuevamente la versión de Python instalada para asegurar el éxito de la operación.

¿Qué ocurre si desinstalamos la versión previa de Python?

Por ningún motivo desinstalaremos la versión de Python 2 preinstalada en el sistema tanto en Mac como en Linux. Si bien es probable que no la utilicemos nunca, si se encuentra preinstalada, con seguridad el sistema operativo la necesita para su normal funcionamiento. Varias versiones de Python pueden coexistir perfectamente en el mismo sistema, así que dejaremos Python 2.x y abriremos la página de descargas de Python para proceder a instalar Python 3.

Un detalle no menor y que se ha mencionado en el paso a paso de instalación para Linux es que varias versiones de Python pueden coexistir en el sistema, por lo tanto, si la versión 2 ya estaba instalada en Mac, ahora se encontrarán tanto la 2 como la 3. Para utilizar el intérprete con la versión 2 de Python, desde la terminal escribiremos:

```
python
```

Ya que por default la versión de Python que se busca es la 2. Si en cambio se quiere utilizar el intérprete con la versión 3, entonces desde la terminal escribiremos:

```
python3
```

De esta forma, utilizaremos la versión que se acaba de instalar según el paso a paso anterior.

INTÉRPRETES DE PYTHON

Como ya hemos mencionado, hay varios intérpretes de Python que se desarrollaron de diferentes maneras, en distintos lenguajes y con variados propósitos. A continuación se mencionan los más destacables.

CPYTHON

Es la implementación oficial y más ampliamente utilizada del lenguaje de programación Python. Cuando instalamos Python, estamos también instalando esta implementación del intérprete. Es decir que tanto desde Windows como desde Linux y Mac, cuando testeamos desde la consola las diferentes pruebas de código que realizamos hasta el momento, sin saberlo hemos utilizado **Cpython**.

Está escrita en C, como podemos suponer por su nombre. Además de CPython, hay otras implementaciones con calidad para producción: **Jython**, escrita en Java; **IronPython**, escrita para el Common Language Runtime, y **PyPy**, escrita en un subconjunto del propio lenguaje Python.

ANACONDA

Anaconda es una distribución libre y abierta de los lenguajes Python y R, que se utiliza en ciencia de datos y machine learning. Anaconda se emplea principalmente para procesamiento de grandes volúmenes de información, análisis predictivo y cómputos científicos.

Las diferentes versiones de los paquetes se administran mediante el sistema de administración del paquete Conda, que lo hace bastante sencillo de instalar, correr, y actualizar software de ciencia de datos y machine learning, tales como **Scikit-team**, **TensorFlow** y **SciPy.3**.

La distribución Anaconda es utilizada por 6 millones de usuarios e incluye más de 250 paquetes de ciencia de datos válidos para Windows, Linux y MacOS.

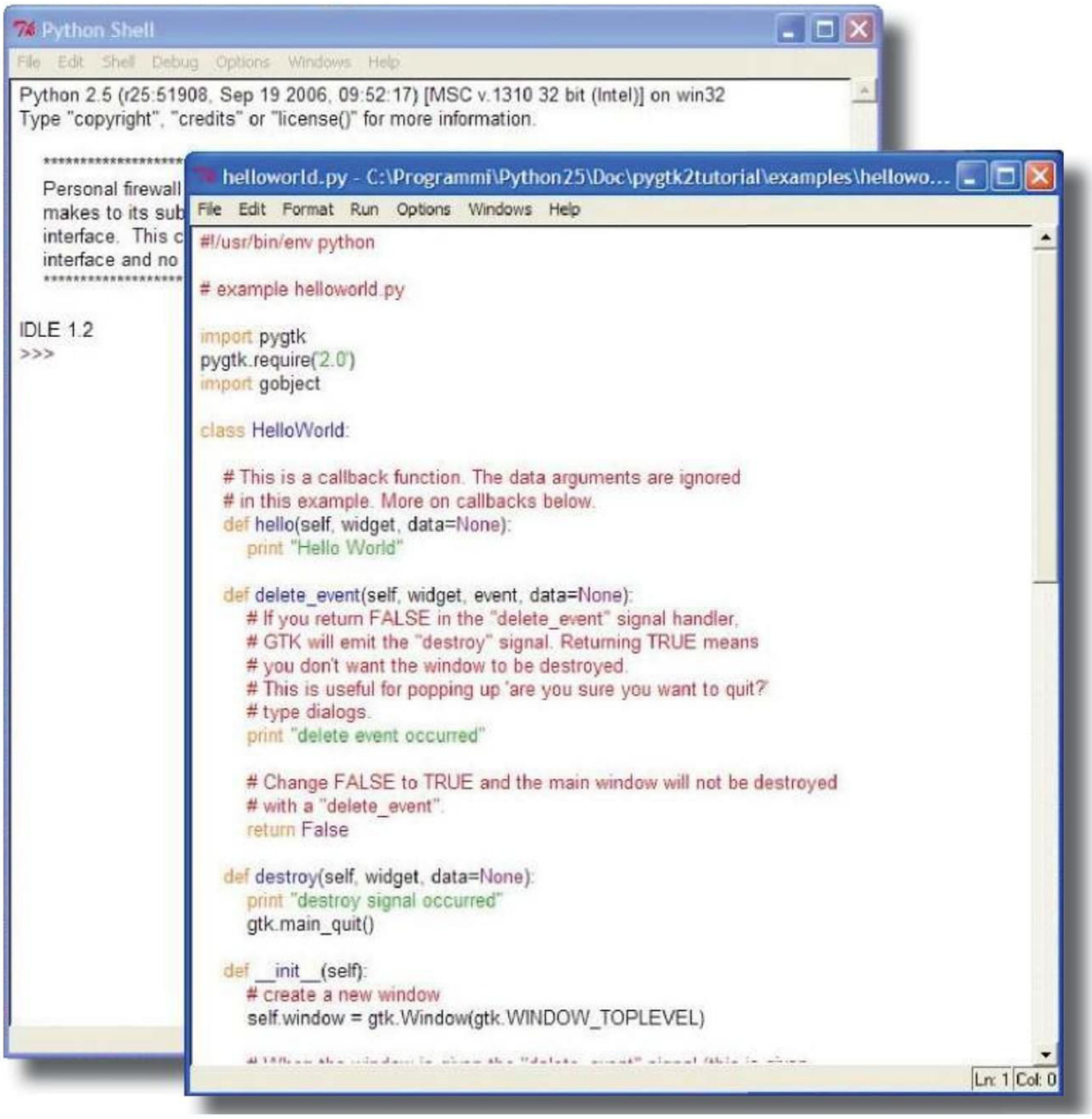
PYPY

Es una implementación de Python escrita en el propio lenguaje Python; esto permite realizar ciertas modificaciones sobre el propio lenguaje y da lugar a los desarrolladores a realizar mejoras y cambios sustanciales sobre el lenguaje. Al estar implementado en un lenguaje de alto nivel PyPy es más flexible y permite mayor experimentación que CPython.

PyPy tiene por objeto proporcionar una traducción común y un framework conceptual para la producción de implementaciones de lenguajes dinámicos, haciendo hincapié en una separación limpia entre la especificación del lenguaje y los aspectos de implementación. Intenta además proporcionar una implementación compatible, flexible y rápida del lenguaje Python utilizando el mencionado framework para desarrollar nuevas características avanzadas sin tener que codificar detalles a bajo nivel.

Sintaxis

BASES DE LA SINTAXIS DE PYTHON



Una de las principales ventajas del uso de Python es su sintaxis clara y simple. En este capítulo recorreremos los fundamentos básicos de la sintaxis de Python y formalizaremos algunos conceptos que nos permitirán comprender el uso del intérprete.

03

Una particularidad de la sintaxis de Python consiste en el llamado *duck typing*, que es el estilo de tipificación de los datos que permite declarar implícitamente el tipo de dato de una variable en el momento de su asignación, accediendo de esta manera que esta variable sea capaz de cambiar de tipo de dato a lo largo de su existencia. Para quienes estén acostumbrados a algún tipo de lenguaje de programación compilado esto es una gran diferencia, ya que para los demás lenguajes la normalidad es tener que definir un tipo de dato para una variable, que morirá con este tipo de dato y no tendrá posibilidad de cambiar.

El duck typing brinda flexibilidad al lenguaje, pero requiere la responsabilidad por parte del programador de recordar con qué tipo de datos se está trabajando a lo largo del código. El nombre del concepto se refiere a la prueba del pato, una humorada de razonamiento inductivo atribuida a James Whitcomb Riley: **"Si veo un ave que nada como pato, suena como pato, camina como pato, lo llamo *pato*"**

REGLAS GENERALES

Para comenzar a escribir código en Python, es preciso comprender algunas reglas generales de la sintaxis del lenguaje. A continuación se listan las principales reglas para tener en cuenta a la hora de realizar un código.

Líneas físicas y líneas lógicas

Podemos entender por **líneas físicas** a aquellas formadas por una secuencia de caracteres que terminan con un carácter de fin de línea (`\n` para sistemas Unix, o `\r\n` para Windows). Por ejemplo:

```
>>>print('Esto es una lineafisica')
Esto es una lineafisica
```

Es decir, en un archivo de texto plano, una línea física terminaría al presionar **ENTER**.

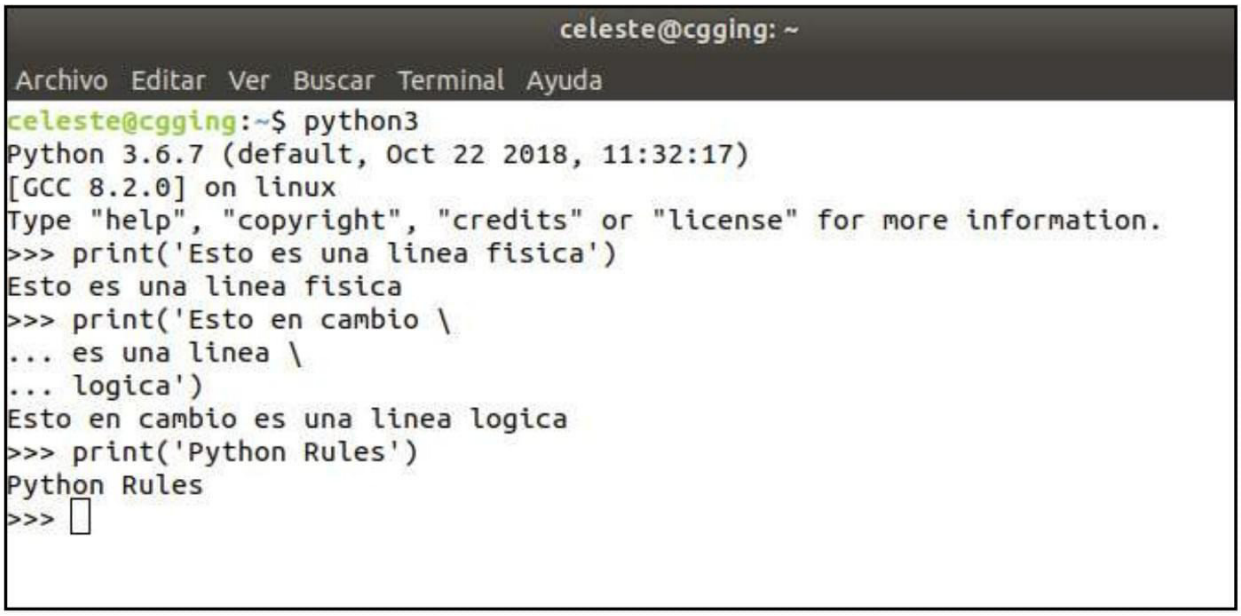
Una **línea lógica**, en cambio, puede estar formada por varias líneas físicas. Hay dos formas de unir líneas físicas para formar una línea lógica, una forma implícita, que consiste en utilizar barra invertida **** colocada en la línea física que se quiere unir, justo antes del carácter de fin de linea; o la forma implícita, que consiste en encerrar las líneas físicas que se quieren unir utilizando los pares de caracteres: **()**, **[]**, **{}**.

Veamos un ejemplo: si una línea lógica se inicia con un paréntesis, se extenderá por tantas líneas físicas como sea necesario y solo terminará con el carácter de cierre. Esto también se cumple para los corchetes **[]** y las llaves **{}**.

La unión implícita es la forma recomendada y generalmente empleada por la mayoría de los programadores con experiencia en Python para poder visualizar el código de manera más cómoda.

Veamos un ejemplo de línea lógica declarada en forma implícita:

```
>>>print(
... 'Hello',
... 'World!'
... )
HelloWorld!
```



CONSTRUCCIÓN DE LAS SENTENCIAS

Habiendo comprendido la diferencia entre líneas físicas y lógicas, solo cabe aclarar que en Python, cuando hablemos de líneas, siempre nos estaremos refiriendo a líneas lógicas, entonces podemos comenzar a hablar sobre **sentencias**.

Una sentencia es una instrucción que el intérprete de Python puede ejecutar.

Las sentencias se pueden construir a partir de una línea física o de una línea lógica, dependiendo de la complejidad y composición de la instrucción que se desea ejecutar.

SENTENCIAS SIMPLES Y COMPUESTAS

Las sentencias de Python se componen de diferente número de líneas lógicas; sabiendo esto, podemos realizar una clasificación sencilla de las sentencias entre simples y compuestas:

Las **sentencias simples** son aquellas que deben completarse en una única línea lógica, como por ejemplo:

```
>>>fromsysimportplatform
```

Las **sentencias compuestas**, en cambio, son aquellas que deben comenzar con una condición de sentencia compuesta y deben contener sentencias simples o compuestas indentadas, a las cuales se las suele llamar *cuerpo* o *bloque*. La condición inicial o encabezado de una sentencia compuesta comienza siempre con una palabra reservada (*keyword*) y termina con el carácter dos puntos (`:`)

Por ejemplo:

```
>>>if a > b:
...    print(a, 'isgreaterthan', b)
```

A diferencia de otros lenguajes, Python no tiene declaraciones u otros elementos sintácticos de alto nivel, solo sentencias, que generalmente ocupan una o varias líneas físicas en el editor que se utilice.

El fin de una línea física, casi siempre determina el fin de la mayoría de las sentencias. Como se mencionó con anterioridad, las líneas físicas terminan con la secuencia de fin de línea **\n** en sistemas Unix o **\r\n** en Windows:

```
>>>var = 'Welcome to Python Scouts!' # Línea física que ter-
mina con la secuencia de fin de línea \n
>>>
```

Otra forma de terminar una sentencia, y esto será familiar para aquellos que hayan tenido contacto con algún lenguaje de programación previo a la lectura de este libro, es emplear el uso del carácter punto y coma (**;**) para terminar las sentencias:

```
>>>print(var); # Sentencia que termina con ;
Welcome to Python Scouts!
```

Otro uso que se le puede dar a **;** y que es el más difundido entre los programadores Python es incluir varias sentencias simples en una misma línea física:

```
>>> var1 = 0; var2 = 1 # Empleo del ; para separar dos sen-
tencias en una misma línea física
>>> var1
0
>>> var2
1
```

Pero, deteniéndonos un segundo en este último uso, no es muy difícil entender que el empleo de **;** para separar varias sentencias en una misma línea física atenta contra la legibilidad del código desarrollado en Python, haciendo engorrosa su interpretación por cualquier otro programador, o por la propia persona que escribió dicho código luego de un tiempo de realizado. Por este motivo el empleo de **;** de esta manera se considera una mala práctica y se recomienda evitarlo.

BUENAS PRÁCTICAS: REGLAMENTO TÁCITO

Cuando empezamos a trabajar con cualquier lenguaje de programación, tenemos que aprender las reglas básicas del lenguaje; si no respetamos la sintaxis, es de esperar que el intérprete o el compilador del lenguaje que estemos utilizando nos dé un error y no podamos ejecutar el código en cuestión.

Sin embargo, en la programación en general se fueron estipulando con los años reglas que se consideran *buenas prácticas* y que ayudan a la legibilidad del código de la misma forma que al trabajo colaborativo.

Por lo general, el código funcionará igual si no respetamos estas buenas prácticas, pero a lo largo de este libro iremos viendo que su uso nos simplificará en gran parte la tarea de programación.

Algunos ejemplos de buenas prácticas que podemos mencionar son:

- **Elegir nombres significativos para las variables.**
- **Evitar la incorporación de más de una instrucción por línea.**
- **Escribir los códigos de la manera más sencilla posible.**
- **Hacer uso del estilo de codificación estándar, ya que nos permite poder mostrar y consultar al respecto de nuestro código en distintas comunidades web de ser necesario.**

A lo largo de esta obra, iremos explayándonos sobre el uso de buenas prácticas.

INDENTACIÓN

El lenguaje Python, a diferencia de otros lenguajes, no emplea llaves o estructuras `begin...end` para definir bloques de código. Para esto, el lenguaje se basa en el uso de lo que se conoce como **indentación**.

La indentación consiste en la inclusión de espacios o caracteres de tabulación al inicio de las líneas lógicas.

Los niveles de indentación corresponden a los distintos bloques del programa.

La indentación del código tiene su origen en la necesidad de hacer que el código sea más legible y comprensible. Esta es la razón fundamental por la cual

Python la incluye en forma directa como parte de su sintaxis, esencialmente para mejorar la legibilidad, comprensión y sencillez del código. Este es uno de los rasgos identificativos y más valorados del lenguaje.

Las sentencias pueden ser agrupadas dentro de una cláusula o cabecera (*header*) de sentencia compuesta mediante la indentación.

De este modo, Python usa la indentación de las líneas lógicas para determinar la agrupación de sentencias y su pertenencia a determinado bloque o cuerpo de sentencia compuesta. Por ejemplo:

```
defprinter():
    # Los espacios al inicio de estas líneas forman la in-
dentación
print('HelloWorld!')
print('Welcome to Python Scouts!')
    # Las sentencias anteriores forman el bloque o cuerpo de
la función printer()

printer() # El llamado a printer() queda fuera del bloque,
pues ya no hay indentación
```

La indentación puede definirse con caracteres de espacio (se recomienda el empleo de cuatro espacios, que es considerado el estilo óptimo de Python) o de tabulación. Es recomendable no mezclar ambos tipos de caracteres en un mismo fragmento de código. De hecho, dependiendo de cómo esté configurado el editor de texto, algunos intérpretes pueden devolver error en este caso. La indentación debe ser la misma (igual cantidad de espacios), al menos para las líneas que componen un mismo bloque de código, y la primera sentencia de un archivo de código no debe tener indentación.

Como se puede observar, la indentación es un componente fundamental en la sintaxis de Python.

Si bien en otros lenguajes de programación la indentación es considerada una buena práctica, en Python es una característica fundamental del lenguaje que, de utilizarse en forma errónea, producirá que el código no pueda ejecutarse, por lo que particularmente en Python no se considera como una buena práctica, sino como una característica obligatoria de la sintaxis.

COMENTARIOS

Los **comentarios** consisten en aclaraciones o indicaciones que resulten de utilidad para el desarrollador o usuario a lo largo del código. En general son secuencias de caracteres que comienzan con el carácter numeral (**#**) y continúan hasta el fin de la línea física. Las líneas físicas que comienzan con **#** son ignoradas completamente por el intérprete de Python. En realidad, estas líneas son muy importantes y están dirigidas a los programadores/mantenedores/clientes del código. En muchas ocasiones seremos nosotros los consumidores de estos comentarios, que nos facilitarán de manera considerable la tarea de comprender, reutilizar o mantener un proyecto, sobre todo si estamos hablando de proyectos grandes o a largo plazo. Los comentarios son útiles para indicar lo que se está realizando en el código y sobre todo por qué se lo hace, no obstante, una buena práctica también consiste en saber cuándo incluir un comentario, y no "ensuciar" el código con comentarios superfluos o redundantes que no aporten información significativa para el proyecto. Por ejemplo:

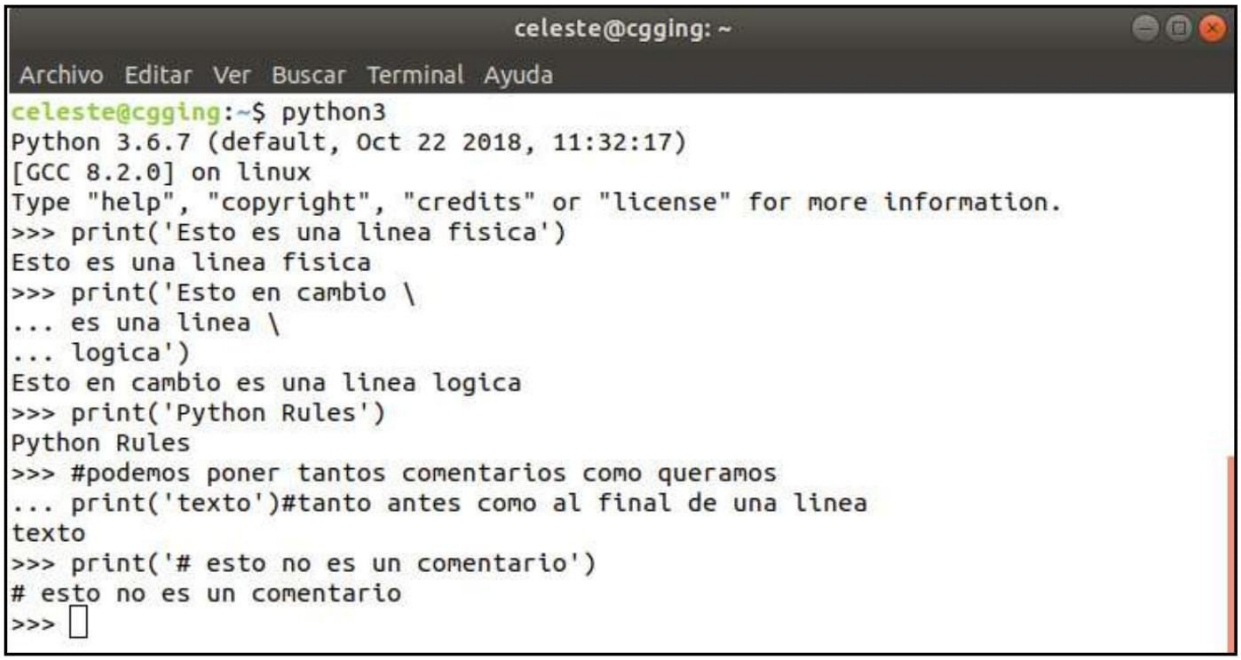
```
>>> # Esto es un comentario que comienza con # y es ignorado por
Python
>>>
>>> # Le asigno el valor 0 a count<= Esto es un comentario innec-
esario
>>>count = 0
>>>
>>> # Inicializo count a 0 para contar las veces que... <= Mejor
>>>count = 0
```

Lo que vimos hasta ahora en el ejemplo son comentarios que se anotan en una línea separada del código. A veces resulta más útil realizar un comentario en línea. Este tipo de comentario se realiza a continuación de la sentencia o línea que se quiere explicar. Veamos un ejemplo:

```
>>>iftemperature<= 273:  # Validar la temperatura <= Comen-
tario en línea
>>>raiseValueError('Wrongtemperaturevalue')
```

Uniendo las líneas físicas en forma implícita, como ya se ha explicado, se pueden incluir comentarios en línea de la siguiente manera:

```
month_names = ['January', 'February', 'March',  # Primer trimes-
tre
               'April', 'May', 'June',  # Segundo trimestre
               'July', 'August', 'September',  # Tercer trimestre
               'October', 'November', 'December']  # Cuarto tri-
mestre
```



TOKENS DEL LENGUAJE

Los **tokens** de Python son componentes fundamentales del lenguaje que forman las líneas lógicas. Ya se ha trabajado con tokens a lo largo de este capítulo, pero no los habíamos mencionado formalmente aún. A modo de ejemplo y para comprender mejor qué es un token, podemos mencionar:

NEWLINE: determina el fin de una línea lógica y el comienzo de otra.
INDENT: indentación de las sentencias dentro de una sentencia compuesta.
DEDENT: fin de indentación que determina el fin de una sentencia compuesta.

Estos tres tokens que se acaban de mencionar fueron utilizados a lo largo de todo el capítulo y se explicó tanto su uso como su importancia. Pero no son los únicos tokens del lenguaje.

Algunos elementos del lenguaje facilitan la creación de distintas estructuras. Entre ellos podemos mencionar el uso de *identificadores*, otro token del lenguaje.

Identifiers o 'identificadores' son aquellos nombres que identifican a variables, funciones, clases, métodos, constantes, módulos, paquetes, etcétera. Dichos tokens comienzan con letras que pueden ser mayúsculas o minúsculas o con guion bajo y luego pueden contener dígitos, letras o más guiones bajos. Es importante tener en cuenta que Python es un lenguaje *case sensitive*, lo que significa que las letras mayúsculas son distintas de las minúsculas.

Por ejemplo:

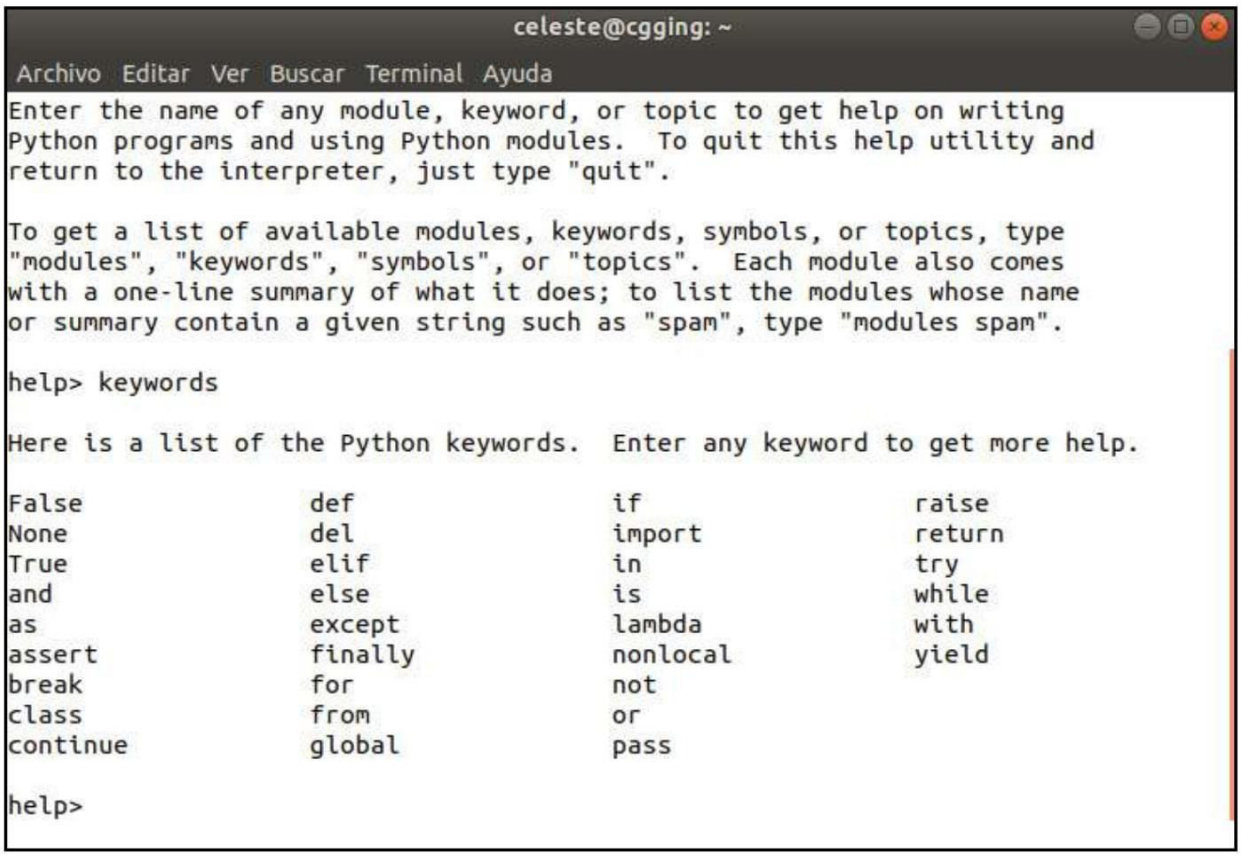
```
>>>a=5 #se declara la variable 'a' y se le asigna el valor
5
>>>A=3 # se declara otra variable independiente de la ante-
rior 'A' y se le asigna el valor 3
>>>a+A
8
```

Espacios y líneas en blanco

Los espacios en blanco pueden ser empleados libremente en el interior de las sentencias (entre tokens), excepto en el inicio de línea, donde los espacios se interpretan como indentación y determinan la pertenencia de una sentencia simple a una compuesta. Las líneas en blanco contienen solo caracteres de espacio, tabulación y fin de línea. Su empleo está estrechamente ligado a la legibilidad del código. Por otro lado, si estamos en una sesión interactiva del intérprete o Ciclo de Lectura, Evaluación, Impresión (REPL por sus siglas en inglés), una línea en blanco representa la conclusión de una sentencia compuesta.

Existen recomendaciones bien establecidas con relación al empleo de espacios y líneas en blanco, y es aconsejable que las sigamos a fin de que nuestro código presente la apariencia de un código Python bien escrito.

Otro token que podemos identificar y que hemos mencionado con anterioridad se refiere a las palabras reservadas (*keywords*): palabras con significado especial para el lenguaje, que no pueden ser empleadas como identificadores. Algunas palabras clave son sentencias simples (por ejemplo, **`break`**, **`continue`**); otras, condiciones de sentencias compuesta (como **`def`**, **`class`**, **`for`**, **`while`**), mientras que otras son operadores (**`and`**, **`or`**, **`is`**, **`in`**). Las palabras reservadas de Python se pueden consultar tecleando en el prompt del intérprete las sentencias siguientes:




```
>>>importkeyword

>>>keyword.kwlist

['False', 'None', 'True', 'and', 'as', 'assert', 'break',
'class', 'continue',
 'def', 'del', 'elif', 'else', 'except', 'finally', 'for',
'from', 'global',
```

```
 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not',
'or', 'pass',
 'raise', 'return', 'try', 'while', 'with', 'yield']
```

También encontramos a los literales o *literals*, valores numéricos o de cadena de caracteres que aparecen directamente escritos en el código. Por ejemplo:

```
>>> 'HelloWorld!' # Literal de string

'HelloWorld!'

r>>> 1452.25 # Literal de float

1452.25

>>> 15 # Literal de int

15

>>> 1_000_000 # Literal de int con guión bajo de agrupación
(versión 3.6)
1000000
```

Otros elementos importantes y que tal vez no nos imaginaríamos que podían clasificarse como token son los operadores u *operators*. Estos son caracteres empleados para denotar operaciones diversas tales como: aritméticas, lógicas, de asignación, etcétera. Los operadores actuales del lenguaje son:

```
+       -       *       **      /       //      %      @

<<>>&       |       ^       ~

<><=      >=      ==      !=
```

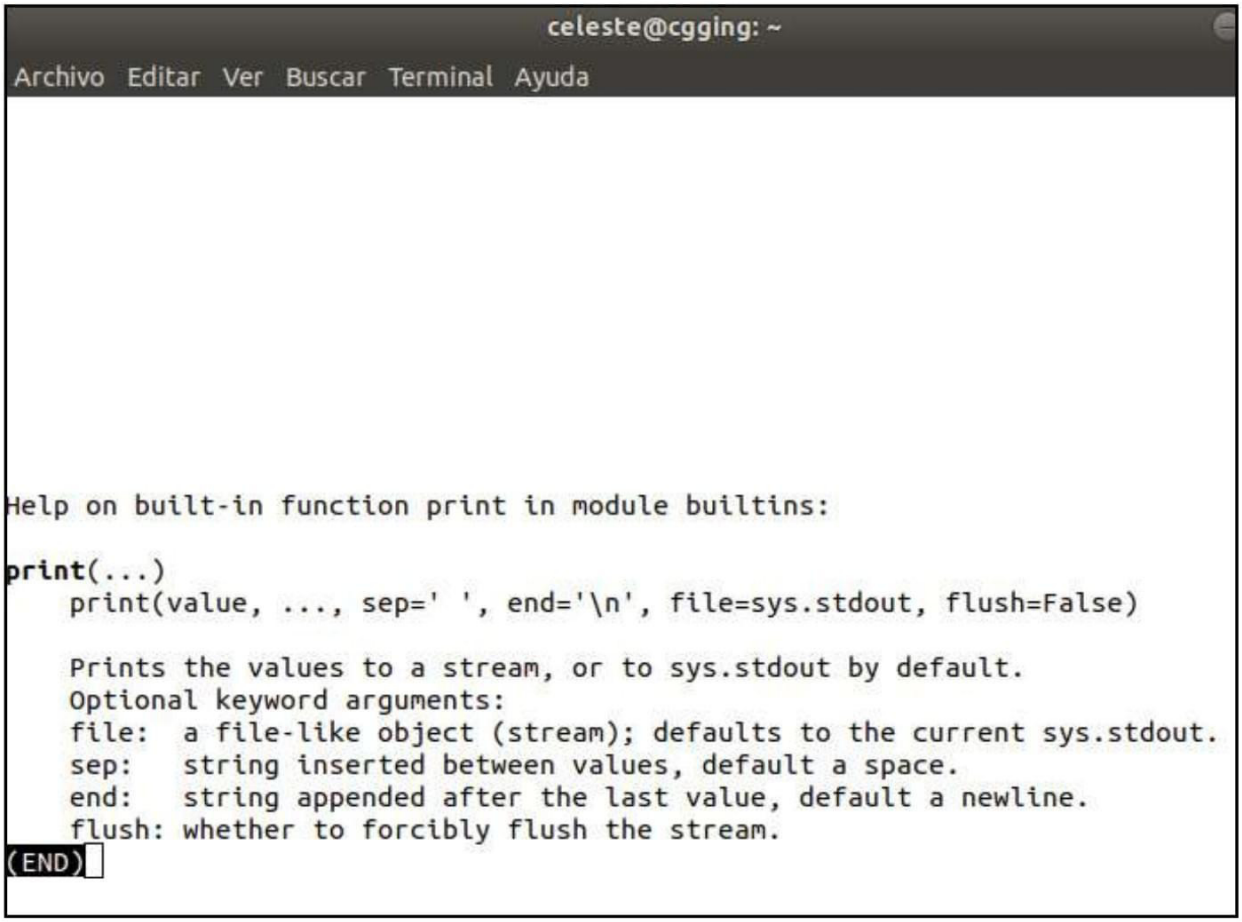
Y finalmente, pero no por eso menos importantes, encontramos a los delimitadores o *delimiters*: caracteres empleados para delimitar literales, líneas lógicas, entre otras. Python incluye los siguientes delimitadores:

```
(       )       [       ]       {       }

,       :       .       ;       @       =       ->

+=      -=      *=      /=      //=     %=      @=

&=      |=      ^=      >>=     <<=     **=
```



PYTHON 2.X VS. PYTHON 3.X

Existen diferencias sintácticas significativas entre las dos ramas actuales del lenguaje, que hacen que el código 2.x no funcione en Python 3.x, y viceversa.

La lista de diferencias es un tanto larga, y de vez en vez se ve aumentada por nuevos elementos. La mejor referencia para mantenerse al día en este tema es la documentación oficial del lenguaje.

Algunas de las diferencias más significativas entre las ramas 2.x y 3.x son:

- **`print` deja de ser sentencia para convertirse en función integrada (built-in).**
- **La captura de excepciones pasa de ser: `except exc, var` a ser `except exc as var`.**
- **El operador de comparación `<>` se elimina en favor de `!=`.**
- **La operación `from module import *` ahora solo se permite a nivel de módulo, eliminando la posibilidad de hacerlo dentro de las funciones**
- **La operación `from .[module] importname` es ahora la única sintaxis aceptada para los import relativos. Todos los import que no comienzan con punto (`.`) son interpretados como absolutos.**
- **La división de enteros (`1 / 2 = 0.5`) retorna números de coma flotante o float (*true division*). El resultado truncado se obtiene ahora con el operador `//` (*floor division*).**

Código

ESCRIBIR CÓDIGO

En los siguientes ejemplos que mostraremos, las entradas y salidas son distinguidas por la presencia o ausencia de los prompts (`>>>` y `...`): para reproducir los ejemplos es necesario escribir todo lo que se encuentre después del prompt, cuando este aparezca; las líneas que no comiencen con el prompt son las salidas del intérprete. Además hay que tener en cuenta que el prompt secundario que aparece por sí solo en una línea de un ejemplo significa que es necesario escribir una línea en blanco; esto es usado para terminar un comando multilínea, como ya hemos mencionado con anterioridad.

Muchos de los ejemplos incluyen comentarios. Ya que los comentarios son para aclarar código y no son interpretados por Python como ya hemos mencionado, pueden omitirse cuando se escriben los ejemplos.

A lo largo de este capítulo empezaremos a escribir código y de esta forma, a familiarizarnos con el uso de Python, en especial mediante la ejecución de diversos ejemplos de códigos. Iremos incrementando la dificultad de estos conforme avancemos.

04

PYTHON EN OPERACIONES MATEMÁTICAS

En forma similar a lo que hemos realizado para probar la correcta instalación de Python en el **capítulo 2**, probaremos algunos comandos simples a partir de un ejemplo. Para eso es necesario iniciar el intérprete.

Números

A lo largo de este apartado, veremos una serie de particularidades del uso de números en Python. Para eso, realizaremos algunos ejemplos básicos utilizando Python como si fuese una calculadora.

El intérprete puede actuar como una calculadora; es decir, es posible ingresar una expresión matemática, y el intérprete nos devolverá los valores resultantes de dicha operación. Intuitivamente ya sabíamos esto luego de los breves testeos que realizamos al instalar Python en nuestro sistema.

La sintaxis es sencilla: los operadores **+**, **-**, ***** y **/** funcionan como en la mayoría de los lenguajes (por ejemplo, Pascal o C); los paréntesis [**()**] pueden ser usados para agrupar. Por ejemplo:

```
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5*6) / 4
5.0
>>> 8 / 5  # la división siempre retorna un número de punto
flotante
1.6
```

Hemos mencionado en capítulos anteriores que Python es un lenguaje "no tipado", esto puede malinterpretarse, pensando que no maneja diferentes tipos de datos, lo cual no es cierto. Python es un lenguaje de alto nivel, por lo que tiene muchas características que le facilitan la actividad al programador. La elección del tipo de dato es una de ellas. Para declarar una variable en Python, basta con escribir su nombre y asignarle un valor. Al realizar este paso, le estamos indicando en forma implícita el tipo de dato que contendrá. No obstante, tenemos el poder de cambiar en forma dinámica el tipo de dato de dicha variable, realizando una nueva asignación que involucre un dato de otra índole.

Veamos cómo maneja Python los tipos de datos numéricos.

Los números enteros (por ejemplo 2, 4, 20) son de tipo **`int`**, aquellos con una parte fraccional (por ejemplo 5.0, 1.6) son de tipo **`float`**.

La división (**`/`**) siempre retorna un punto flotante. Si nos interesa quedarnos solo con la parte entera de una división, lo que se conoce como *floor division*, podemos utilizar el operador **`//`**; si lo que queremos es calcular el resto, utilizaremos **`%`**:

```
>>> 17 / 3  # la división clásica retorna un punto flotante
5.666666666666667
>>>
>>> 17 // 3  # la división entera descarta la parte fraccio-
nal
5
>>> 17 % 3  # el operando % retorna el resto de la división
2
>>> 5 * 3 + 2  # resultado * divisor + resto
17
```

Con Python, es posible usar el operador **`**`** para calcular potencias:

```
>>> 5 ** 2  # 5 al cuadrado
25
>>> 2 ** 7  # 2 a la potencia de 7
128
```

El signo igual (=) es usado para asignar un valor a una variable. Por consiguiente, no se mostrará ningún resultado antes del próximo prompt:

```
>>> ancho = 20
>>> largo = 5 * 9
>>> ancho * largo
900
```

Si una variable no está "definida" (con un valor asignado), intentar usarla producirá un error:

```
>>> n  # tratamos de acceder a una variable no definida
Traceback (mostrecentcalllast):
  File "<stdin>", line 1, in <module>
NameError: name 'n' isnotdefined
```

Hay soporte completo de punto flotante; operadores con operando mezclados convertirán los enteros a punto flotante:

```
>>> 4 * 3.75 - 1
14.0
```

En el modo interactivo, la última expresión impresa es asignada a la variable _. O sea que podemos seguir calculando a partir del último resultado obtenido teniendo en cuenta esa variable:

```
>>> impuesto = 12.5 / 100
>>> precio = 100.50
>>> precio * impuesto
12.5625
```

```
>>> precio + _
113.0625
>>> round(_, 2)
113.06
```

Esta variable debería tomarse como variable de solo lectura. Si bien es posible asignarle un valor, esto no es recomendable, ya que en realidad al asignarle un valor, no estaríamos utilizando dicha variable, sino creando otra variable local con el mismo nombre, enmascarando la variable que mencionamos inicialmente.

Además de `int` y `float`, Python maneja otros tipos de datos numéricos tales como `Decimal` y `Fraction`. Python también tiene soporte integrado para números complejos, y usa el sufijo `j` o `J` para indicar la parte imaginaria (por ejemplo 3+5j).

OPERACIONES MÁS COMPLEJAS CON NÚMEROS

Hasta aquí hemos realizado algunas operaciones con números muy sencillas, con los mismos operandos que utilizaríamos en una calculadora científica y apenas algunos más específicos, tales como quedarnos con la parte entera de un resultado en una división.

Pero ¿qué ocurriría si, de repente, quisiéramos calcular un resultado y sobre esa base tomar una cierta decisión?

Con las herramientas que conocemos hasta el momento, no podríamos realizarlo, por ese motivo introduciremos una serie de herramientas para control de flujo que nos permitirán realizar esto y mucho más.

La sentencia if

Supongamos que queremos realizar unos cálculos cuyo resultado no puede ser nunca negativo, es decir, el resultado no nos interesa si es menor a cero.

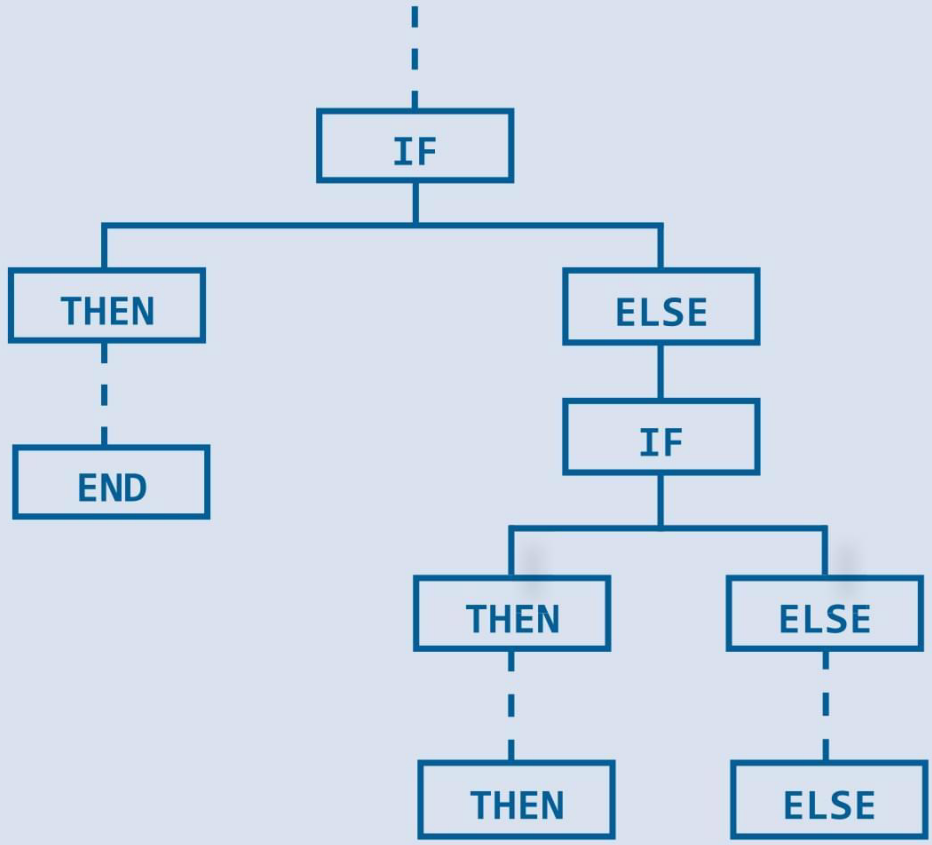
En ese caso, si pensáramos los pasos del código en forma de una lista, para poder trabajar con dicha condición la serie de pasos por seguir sería la siguiente:

1. **Introducir los valores para operar.**

2. **Calcular la operación.**

3. **¿El resultado de la operación es menor a cero?**
   a. Si es menor a cero: mostrar un cartel por pantalla comunicando el error.
   b. Si no es menor a cero: mostrar el resultado de la operación por pantalla.

4. **Fin.**



Lo que acabamos de resumir en unas líneas de texto podría definirse como pseudocódigo, y es de suma importancia y utilidad a la hora de diagramar nuestros programas.

¿Cómo se traduciría esto en lenguaje Python? Introduciendo la sentencia **if**.

La sentencia **if** nos da la posibilidad de plantear un interrogante y continuar el hilo del programa hacia la opción que cumpla con la condición que se ha planteado; para ello, veremos en el siguiente ejemplo tanto el uso de la sentencia **if** como la importancia de la indentación para definir los bloques de acción.

if viene acompañada de otra sentencia: **else**, que actuará en el caso en que la condición que planteamos en **if** no se cumpla. En español sería: "Si pasa esto, entonces actúo de esta manera; si no se cumple, entonces actúo de esta otra manera".

El uso de **else** no es obligatorio, dependerá de la condición que estemos analizando, como veremos en otros ejemplos.

```
>>> valor1=4
>>> valor2=2
>>>result= valor1-valor2
2
>>>if result<0:
…       print('El resultado es negativo')
…   else:
…       print(result)
2
```

¿Qué ocurre si la decisión que tenemos que tomar no es binaria? No siempre nos alcanzará con dos opciones para definir una situación. Supongamos que en el código anterior nuestro resultado solo es útil si el número es mayor a **0** y menor a **100**. En ese caso, introduciremos la sentencia **elif**, esta palabra reservada es una abreviación de **elseif** y se utiliza para simplificar el código.

```
>>> valor1=4
>>> valor2=2
>>>result= valor1-valor2
2
>>>if result<0:
```

```
…       print('El resultado es negativo')
…   elif result>100:
…       print('El resultado se encuentra fuera de rango')
…  else:
....      print(result)
```

En Python se utiliza esta secuencia de **if**/**elif**/**else**, que reemplaza perfectamente a las sentencias **switch**/**case**, que podemos encontrar en otros lenguajes de programación.

La sentencia while

Otra sentencia de suma utilidad y que sin duda necesitaremos en nuestros códigos es la sentencia **while**.

Esta sentencia se utiliza para realizar una repetición de una acción mientras se cumpla con una condición explicitada.

Supongamos que queremos sumar todos los valores entre **1** y **10**:

```
>>>b=1
>>>while b<10:
…       result=result+b
…       b=b+1
… print(result)
```

Analicemos un poco este código: la sentencia **while** genera un bucle de código que se ejecutará mientras la condición se cumpla, en este caso mientras **b** valga menos de **10**.

El bucle en cuestión estará conformado por las líneas que se encuentran indentadas de la misma manera, y en este caso vemos que son dos. Luego se imprime una única vez el resultado, ya que la indentación de dicho print nos da cuenta de que esa línea de código no pertenece al bloque **while** y se ejecutará recién una vez finalizada su ejecución.

CADENAS DE CARACTERES

Hasta el momento hemos visto cómo utilizar variables numéricas y algunas sentencias útiles para aplicar a nuestros códigos y poder realizar tareas más interesantes que cálculos que podríamos realizar con una calculadora.

Veamos ahora otro tipo de dato, que nos permitirá ampliar el horizonte de posibilidades en el mundo de Python.

Las **cadenas de caracteres** en su forma más sencilla son vectores de letras que definimos para formar un texto. Podemos utilizarlas para almacenar mensajes que nos sirvan para darle información al usuario de nuestro código o para almacenar datos que necesitemos procesar de alguna manera.

Las cadenas de texto pueden expresarse de diferentes maneras. Una forma de definirlas es encerrarlas entre comillas simples o dobles.

Existen algunos caracteres especiales que no podrán ingresarse de manera directa en la cadena, y para ellos será necesario contar con algún elemento que nos permita "escapar" de las comillas para poder introducir el carácter en cuestión. Para esos casos, se suele utilizar `\`.

Un ejemplo de cómo definir una cadena de caracteres:

```
>>> 'huevos y pan'  # comillas simples
'huevos y pan'
>>> 'doesn\'t'  # usa \' para escapar comillas simples…
```

```
“doesn’t”
>>> “doesn’t”  # ...o de lo contrario usa comillas doblas
“doesn’t”
>>> ‘”Si,” le dijo.’
‘”Si,” le dijo.’
>>> “\”Si,\” le dijo.”
‘”Si,” le dijo.’
>>> ‘”Isn\’t,” shesaid.’
‘”Isn\’t,” shesaid.’
```

Analicemos la segunda sentencia del ejemplo anterior. Si no utilizáramos **** para escapar de las comillas simples, el intérprete entendería que la cadena de caracteres termina en el apóstrofo del texto, y lo demás generaría un error al no entenderse como parte de la cadena.

Para entender mejor el texto que estamos introduciendo en la cadena, podemos valernos de la función **print()** y obtener una salida más legible, ya que nos mostrará el texto de la cadena tal cual lo vería el usuario final, sin las comillas ni los escapes para caracteres especiales:

```
>>> ‘”Isn\’t,” shesaid.’
‘”Isn\’t,” shesaid.’
>>>print(‘”Isn\’t,” shesaid.’)
“Isn’t,” shesaid.
>>> s = ‘Primera línea.\nSegunda línea.’  # \n significa
nueva línea
>>> s  # sin print(), \n es incluído en la salida
‘Primera línea.\nSegunda línea.’
>>>print(s)  # con print(), \n produce una nueva línea
Primera línea.
Segunda línea.
```

Si no queremos que los caracteres antepuestos por **** sean interpretados como caracteres especiales, podemos usar cadenas crudas agregando una **r** antes de la primera comilla:

```
>>>print('C:\algun\nombre')  # aquí \n significa nueva línea!
C:\algun
ombre
>>>print(r'C:\algun\nombre')  # nota la r antes de la comilla
C:\algun\nombre
```

Podemos escribir cadenas de texto con múltiples líneas. Para eso se suele utilizar triple comilla para encerrar al texto, pero también pueden utilizarse comillas dobles o simples indistintamente.

Los finales de línea se agregan en forma automática, aunque es posible evitarlos incorporando **** al final de la línea. Veamos un ejemplo:

```
print("""\
Uso: algo [OPTIONS]
     -h                    Muestra el mensaje de uso
     -H nombrehost         Nombre del host al cual conectarse
""")
```

Produce la siguiente salida (notemos que la línea inicial no está incluida):

```
Uso: algo [OPTIONS]
     -h                    Muestra el mensaje de uso
     -H nombrehost         Nombre del host al cual conectarse
```

Una operación útil que podemos realizar al tener diferentes cadenas de caracteres es la concatenación; esta consiste en pegar una cadena de caracteres a continuación de otra y se logra con el operador **+**. Además de concatenar, también es posible repetir una cadena de caracteres con el operador *****:

```
>>> # 3 veces 'un', seguido de 'ium'
>>> 3 * 'un' + 'ium'
'unununium'
```

Como podemos notar en el ejemplo anterior, debemos indicar cuántas veces queremos repetir o multiplicar la cadena de texto previo al uso del operador *****.

Una particularidad de las cadenas literales, es decir, aquellas que definimos encerrando entre comillas, es que si las colocamos una al lado de otra, se concatenaran automáticamente, una característica que solo funciona con dos y solo dos literales, y no sirve para concatenar con variables o expresiones:

```
>>> 'Py' 'thon'
'Python'
```

En el ejemplo anterior vimos lo que ocurre con dos literales, ahora veamos qué pasa si queremos utilizar este método con una variable y una cadena literal:

```
>>>prefix = 'Py'
>>>prefix 'thon'  # no se puede concatenar una variable y
una cadena literal
  ...
SyntaxError: invalidsyntax
>>> ('un' * 3) 'ium'
  ...
SyntaxError: invalidsyntax
```

En este caso, para lograr la concatenación, la solución es usar el operador **+**:

```
>>>prefix + 'thon'
'Python'
```

Una particularidad muy útil de las cadenas de texto es que se pueden indexar, el primer carácter de la cadena tiene el índice **0**. No hay un tipo de dato para los caracteres; un carácter es simplemente una cadena de longitud uno:

```
>>> palabra = 'Python'
>>> palabra[0]  # caracter en la posición 0
'P'
>>> palabra[5]  # caracter en la posición 5
'n'
```

Algo que Python nos permite realizar a diferencia de otros lenguajes y que resulta de mucha utilidad es utilizar índices negativos, y de esta manera comenzar a contar de atrás para adelante en una cadena de caracteres:

```
>>> palabra[-1]  # último caracter
'n'
>>> palabra[-2]  # ante último caracter
'o'
>>> palabra[-6]
'P'
```

Es posible obtener subcadenas utilizando índices, para ello simplemente escribiremos los índices entre los cuales se contiene la subcadena que nos interesa, de la siguiente manera:

```
>>> palabra[0:2]  # caracteres desde la posición 0 (incluída)
hasta la 2 (excluída)
'Py'
>>> palabra[2:5]  # caracteres desde la posición 2 (incluída)
hasta la 5 (excluída)
'tho'
```

El primer carácter que denotamos siempre es incluido, mientras que el último siempre se excluye. De esta forma si pusiéramos **`s[:i]+s[i:]`**, la concatenación nos dará la cadena s sin agregado ni omisión de caracteres:

```
>>> palabra[:2] + palabra[2:]
'Python'
>>> palabra[:4] + palabra[4:]
'Python'
```

Los índices de las subcadenas tienen valores por defecto útiles; el valor por defecto para el primer índice es **0**, el valor por defecto para el segundo índice es la longitud de la subcadena por extraer.

```
>>> palabra[:2]  # caracteres desde el principio hasta la
posición 2 (excluída)
'Py'
>>> palabra[4:]  # caracteres desde la posición 4 (incluí-
da) hasta el final
'on'
>>> palabra[-2:] # caracteres desde la ante-última (incluí-
da) hasta el final
'on'
```

Una forma de recordar cómo funcionan las extracciones de subcadenas es pensar en los índices como puntos entre caracteres, con el punto a la izquierda del primer carácter numerado en **0**. Luego, el punto a la derecha del último carácter de una cadena de **n** caracteres tienen índice n, por ejemplo:

```
+---+---+---+---+---+---+
 | P | y | t | h | o | n |
 +---+---+---+---+---+---+
 0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

Como podemos observar, la primera fila de números nos da la posición de comienzo a fin con los números **0** a **6**. La segunda fila en cambio nos da los índices negativos para poder llamar a cada posición desde el último hasta el primer carácter.

Para índices no negativos, la longitud de la subcadena es la diferencia de los índices, si ambos entran en los límites. Por ejemplo, la longitud de palabra **[1:3]** es **2**.

Si no sabemos la longitud de una cadena, podemos suponer que poniendo un índice muy grande obtendremos el texto completo, pero no es así. Usar un índice mayor a la longitud de la cadena producirá un error:

```
>>> palabra[42]  # la palabra solo tiene 6 caracteres
Traceback (mostrecentcalllast):
  File “<stdin>”, line 1, in <module>
IndexError: stringindexout of range

No obstante, esto no sucede en las subcadenas:

>>> palabra[4:42]
‘on’
>>> palabra[42:]
‘’
```

Otra característica de las cadenas de caracteres en Python es que no pueden ser modificadas, es decir que una vez que fueron definidas, sus elementos permanecen inmutables. Por eso, asignar a una posición indexada de la cadena resulta en un error:

```
>>> palabra[0] = ‘J’
  ...
TypeError: ‘str’ objectdoesnotsupportitemassignment
>>> palabra[2:] = ‘py’
  ...
TypeError: ‘str’ objectdoesnotsupportitemassignment
```

Para utilizar una cadena diferente a la creada, es necesario crear otra cadena:

```
>>> 'J' + palabra[1:]
'Jython'
>>> palabra[:2] + 'py'
'Pypy'
```

Para solucionar el problema de no conocer la longitud de la cadena de texto y poder utilizar todos sus elementos para trabajar, contamos con la función **`len()`**, que nos devuelve la longitud de la cadena de texto:

```
>>> s = 'supercalifrastilisticoespialidoso'
>>>len(s)
33
```

MAS SENTENCIAS ÚTILES

Ahora que ya sabemos cómo podemos utilizar otros tipos de datos además de los numéricos, es posible investigar un poco más sobre las sentencias que tenemos disponibles para hacer códigos interesantes en Python.

Listas

Python tiene varios tipos de datos compuestos, usados para agrupar otros valores. El más versátil es lista; está puede ser escrita como una lista de valores separados por coma (ítems) entre corchetes. Las listas pueden contener ítems de diferentes tipos, pero usualmente los ítems son del mismo tipo:

>>> cuadrados = [1, 4, 9, 16, 25]
>>> cuadrados [1, 4, 9, 16, 25]

La sentencia for

En otros lenguajes de programación, hubiéramos explicado la sentencia **for** a la par de la sentencia **while**, ya que nos permitiría recorrer una serie de datos o realizar cálculos con un principio, un fin y un cierto paso. Pero en Python la sentencia **for** es diferente, itera sobre los ítems de cualquier secuencia (una lista o una cadena de texto) en el orden que aparecen en la secuencia.
Por ejemplo:

```
>>> # Midiendo cadenas de texto
... palabras = [‘gato’, ‘ventana’, ‘defenestrado’]
>>>for p in palabras:
...       print(p, len(p))
...
gato 4
ventana 7
defenestrado 12
```

Si necesitáramos modificar la secuencia sobre la que se está iterando mientras estamos dentro del ciclo (por ejemplo para borrar algunos ítems), se recomienda primero realizar una copia. Iterar sobre una secuencia no hace de manera implícita una copia. La notación de subcadena es especialmente conveniente para esto:

```
>>>for p in palabras[:]:  # hace una copia por subcadena de
toda la lista
...       iflen(p) > 6:
...           palabras.insert(0, p)
...
>>> palabras
[‘defenestrado’, ‘ventana’, ‘gato’, ‘ventana’, ‘defenestrado’]
```

Con **for w in words:**, el ejemplo intentaría crear una lista infinita.

La función range()

Antes mencionamos que la sentencia **for** se utiliza de manera diferente en Python que en otros lenguajes, no obstante, si necesitáramos iterar sobre una secuencia de números, contamos con la función integrada **range()**, que genera progresiones aritméticas y podríamos utilizarla en combinación con **for:**.

```
>>>for i in range(5):
...         print(i)
...
0
1
2
3
4
```

range(5) produce cinco valores comenzando en **0**, es decir **0**, **1**, **2**, **3** y **4**. El valor final no es parte de la secuencia. Es posible realizar ciertas especificaciones sobre la función **range()**, tales como que comience con otro valor diferente de **0** o que incremente con una cantidad particular en vez de realizar incrementos de **1**, incluso es posible configurarlo en decremento:

```
range(5, 10)
   5 through 9

range(0, 10, 3)
   0, 3, 6, 9

range(-10, -100, -30)
  -10, -40, -70
```

Para iterar sobre los índices de una secuencia, es posible utilizar **range()** y **len()** en combinación:

```
>>> a = [‘Mary’, ‘tenia’, ‘un’, ‘corderito’]
>>>for i in range(len(a)):
...         print(i, a[i])
...
0 Mary
1 tenia
2 un
3 corderito
```

Si queremos realizar un print de un range para mostrar los resultados por pantalla, nos devuelve algo curioso:

```
>>>print(range(10))
range(0, 10)
```

No debemos confundir el objeto que nos devuelve **`range()`** con una lista; si bien se ve de forma similar, en realidad, es un objeto que devuelve los ítems sucesivos de la secuencia deseada, pero no se construye la lista y de esa manera se ahorra espacio.

Entonces, es un objeto iterable; esto es, que se lo puede usar en funciones y construcciones que esperan algo de lo cual obtener ítems sucesivos hasta que se termine. Si queremos construir una lista, contamos con la función **`list()`**, que creará listas a partir de objetos iterables:

```
>>>list(range(5))
[0, 1, 2, 3, 4]
```

LAS SENTENCIAS BREAK, CONTINUE Y ELSE EN LAZOS

La sentencia **break,** como en C, termina el lazo **for** o **while** más anidado. Las sentencias de lazo pueden tener una cláusula **else** que es ejecutada cuando el lazo termina, luego de agotar la lista (con **for**) o cuando la condición se hace falsa (con **while**), pero no cuando el lazo es terminado con la sentencia **break**. Se ejemplifica en el siguiente lazo, que busca números primos:

```
>>>for n in range(2, 10):
...         for x in range(2, n):
...             if n % x == 0:
...                 print(n, 'es igual a', x, '*', n/x)
...                 break
...         else:
...             # sigue el bucle sin encontrar un factor
...             print(n, 'es un numero primo')
...
2 es un numero primo
3 es un numero primo
4 es igual a 2 * 2
5 es un numero primo
6 es igual a 2 * 3
7 es un numero primo
8 es igual a 2 * 4
9 es igual a 3 * 3
```

Observemos el detalle de que el **else** no pertenece al **if**, sino al ciclo **for**. Esto podemos observarlo fácilmente por la indentación.

Cuando se usa con un ciclo, el **else** tiene más en común con el **else** de una declaración **try** que con el de un **if**: el **else** de un **try** se ejecuta cuando no se genera ninguna excepción, y el **else** de un ciclo se ejecuta cuando no hay ningún **break**.

La declaración **continue**, también tomada de C, continúa con la siguiente iteración del ciclo:

```
>>>for num in range(2, 10):
...         if num % 2 == 0:
...             print("Encontré un número par", num)
...             continue
...         print("Encontré un número", num)
Encontré un número par 2
Encontré un número 3
Encontré un número par 4
Encontré un número 5
Encontré un número par 6
Encontré un número 7
Encontré un número par 8
Encontré un número 9
```

Errores y excepciones en el código

Cuando comenzamos a escribir código, somos más propensos a cometer algunos errores de sintaxis y, considerando que cometer errores es muy común en programación aún en los programadores más experimentados, vale mencionar los dos tipos más frecuentes de errores con los que nos encontraremos.

Es posible dividir los errores en errores de *sintaxis y excepciones*. Podemos identificar los errores de sintaxis por haber cometido algún error al escribir una estructura, omitir algún signo de puntuación, etcétera. Las excepciones en cambio son errores que se producen en tiempo de ejecución, por ejemplo, si en nuestro código intentamos acceder a un archivo de texto y este no existe en la ubicación especificada, se producirá una excepción.

LA SENTENCIA PASS

La sentencia **pass** no hace nada. Se puede usar cuando una sentencia es requerida por la sintaxis, pero el programa no requiere ninguna acción. Por ejemplo:

```
>>>while True:
...    pass  # Espera ocupada hasta una interrupción de
teclado (Ctrl+C)
...

Se usa normalmente para crear clases en su mínima expre-
sión:
>>>class MyEmptyClass:
...    pass
 ...
```

Otro lugar donde se puede usar **pass** es como una marca de lugar para una función o un cuerpo condicional cuando estamos trabajando en código nuevo; esto nos permite testear el código sin tener que implementar esa parte de código antes de hacer el testeo. El **pass** se ignora silenciosamente:

```
>>>def initlog(*args):
...    pass   # Acordate de implementar esto!
...
```

Ejemplos Prácticos

EJECUTAR DESDE UN ARCHIVO

A continuación, realizaremos varios ejemplos prácticos que nos servirán para ir agilizando nuestra codificación en Python.

Antes de comenzar, es útil saber cómo realizar la codificación desde un archivo de texto y luego llamarla desde el intérprete.

Los programas que realizamos pueden ser ejecutados directamente desde un archivo de texto almacenado en nuestro sistema.

Si queremos realizar los programas en un archivo, estos deben ser de texto plan;, respetar la estructura de código Python, es decir, la indentación fundamentalmente; y almacenarse con la extensión `.py`, por ejemplo: `holaMundo.py`.

Para ejecutar dicho código directamente desde un archivo en vez de copiarlo al intérprete interactivo, lo que tenemos que hacer es ingresar al directorio en donde se encuentra almacenado dicho archivo y escribir la siguiente línea:

```
python3 holaMundo.py
```

Esto ejecutará directamente el código sin mostrarnos todas las líneas que escribimos, a las que podremos editar, de ser necesario, directamente desde el archivo que creamos.

HOLA MUNDO

El primer y más básico ejemplo práctico que podemos mencionar y que ya hemos probado es el famoso y nunca bien ponderado “Hola Mundo”.

En este capítulo veremos una serie de códigos de ejemplo con diferente nivel de complejidad e introduciremos el concepto de función, que nos resultará sumamente práctico al ir creciendo en la dimensión de nuestros códigos.

05

Para darle cierto aire de novedad a este código, vamos a escribirlo en un documento. Para ello en la consola de Linux, crearemos un nuevo archivo de la siguiente manera:

```
gedit holaMundo.py
```

Esto nos creará, en el caso de que no exista, y abrirá en modo de edición el archivo **holaMundo.py**. Una vez dentro del archivo, procederemos a escribir las siguientes líneas de código:

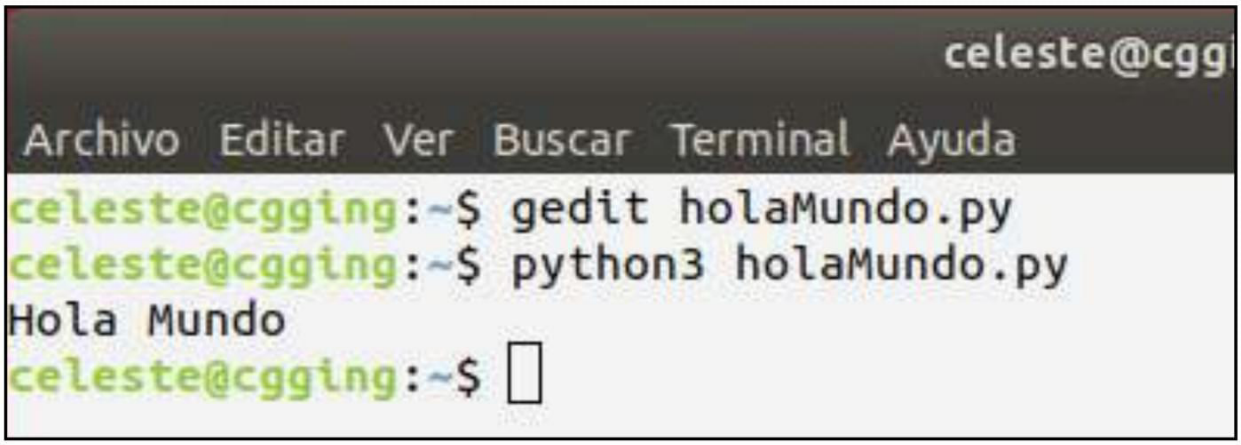
```
print(‘Hola Mundo’)
```

Luego, procederemos a guardar y cerrar el archivo.

Para ejecutar el código que acabamos de escribir, simplemente basta con escribir desde la consola:

```
python3 holaMundo.py
```

Esto producirá la ejecución del código y nos mostrará en la consola los resultados.



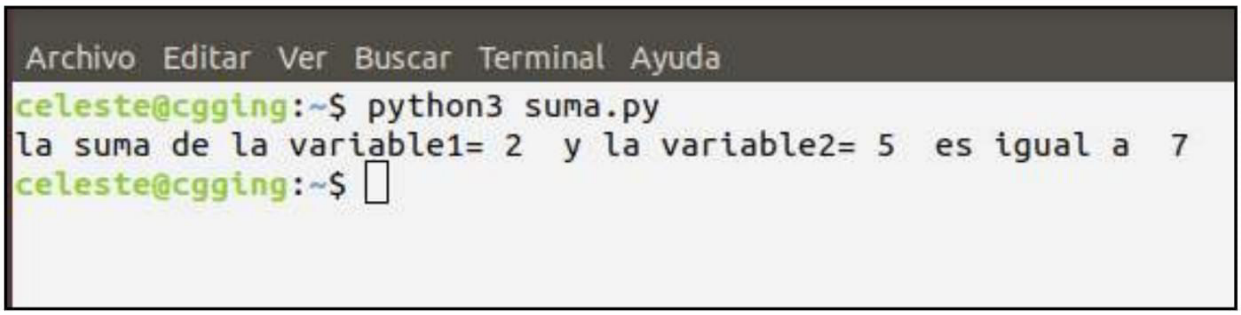
Si bien este ejemplo es sumamente básico, nos sirve para practicar la edición de un archivo de código y su ejecución directamente desde la consola en lugar de trabajar solo con la consola interactiva de Python.

SUMA

En el ejemplo que presentamos a continuación, inicializaremos dos variables, las sumaremos y mostraremos el resultado por pantalla:

```
variable1=3
variable2=5
suma= variable1 + variable2
print("la suma de la variable 1=" , variable1, " y la vari-
able 2=", variable2, "es igual a ", suma)
```

Al ejecutar el código del archivo que acabamos de generar, observaremos lo siguiente:

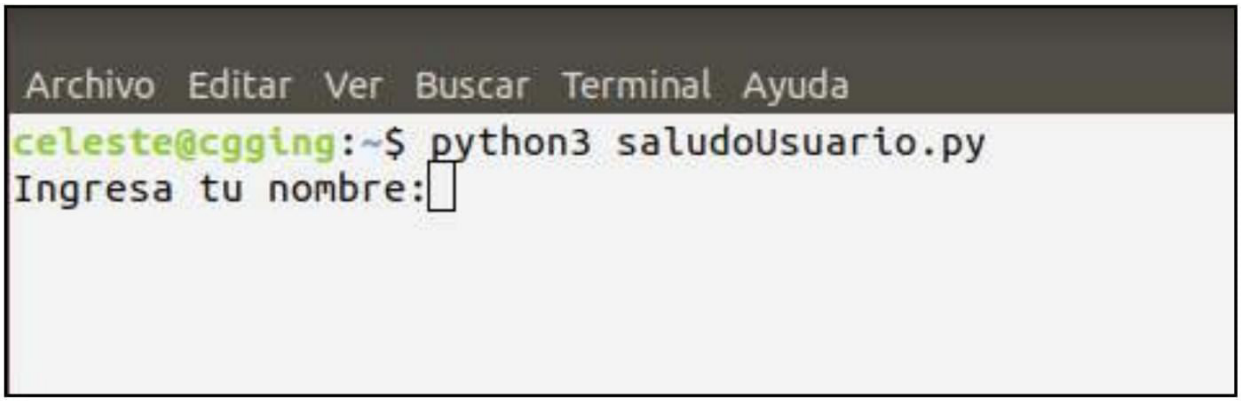


SALUDAR AL USUARIO

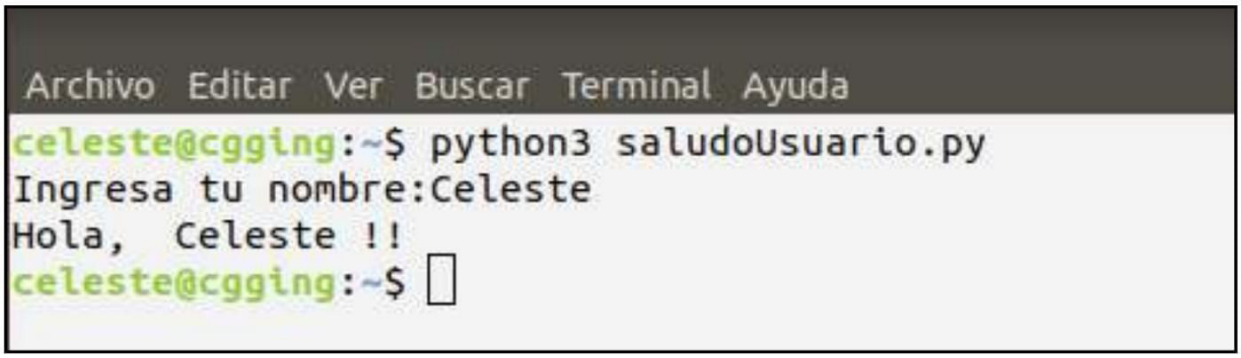
Algo que nos resultará de suma importancia cuando programemos es la interacción con el usuario; habrá ocasiones en las cuales el usuario deberá proporcionar datos al programa para que este pueda ejecutarse. Normalmente la proporción de datos se produce mediante el teclado, y para poder recibirlos contamos con la función **`input()`** en Python:

```
nombre= input("Ingresa tu nombre:")
print("Hola, ",nombre,"!!")
```

Al ejecutar el código anterior, en una primera etapa se espera la interacción del usuario:



Para luego completar la ejecución del código:



SUMA AVANZADA

En el ejemplo anterior vimos cómo ingresar una cadena de texto por teclado, lo cual podría despertar el interrogante sobre si es posible ingresar números por teclado. Frente a esta pregunta, la respuesta es positiva. Todos los datos que ingresamos por teclado utilizando la función **`input()`** se interpretan como cadenas de texto, pero es posible convertirlos utilizando ciertos modificadores.

Para modificar el ejemplo de suma pidiendo el ingreso de datos por teclado, basta con agregar el modificador **`int()`** encerrando a **`input()`**, y de esta forma forzaremos a utilizar solo datos numéricos enteros, si quisiéramos utilizar flotantes, bastaría con reemplazar **`int()`** con **`float()`**. ¿Qué ocurrirá si el usuario intenta ingresar una letra por teclado cuando el programa espera un dato numérico? Simplemente se producirá un error en tiempo de ejecución y no se continuará con el flujo del programa, esto mismo pasaría si no hiciéramos la conversión de datos en el **`input()`**, aun ingresando un dato numérico. Pero veamos todo esto por partes:

El código con la modificación para pedir el ingreso de datos por teclado quedaría de la siguiente manera:

```
variable1=int(input("Ingrese el primer valor:"))
variable2=int(input("Ingrese el segundo valor:"))
suma= variable1 + variable2
print("la suma de la variable 1=" , variable1, " y la vari-
able 2=", variable2, "es igual a ", suma)
```

Con la modificación anterior, al ejecutar el programa y si cargamos datos numéricos, todo funcionará sin problemas:

```
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
celeste@cgging:~$ gedit sumaVitaminas.py
celeste@cgging:~$ python3 sumaVitaminas.py
Ingrese el primer valor:3
Ingrese el segundo valor:4
la suma de la variable 1= 3  y la variable 2= 4 es igual a  7
celeste@cgging:~$
```

Probemos lo que ocurriría si, en vez de ingresar datos numéricos, decidimos ingresar una letra:

```
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
celeste@cgging:~$ python3 sumaVitaminas.py
Ingrese el primer valor:e
Traceback (most recent call last):
  File "sumaVitaminas.py", line 1, in <module>
    variable1=int(input("Ingrese el primer valor:"))
ValueError: invalid literal for int() with base 10: 'e'
celeste@cgging:~$
```

Como podemos ver en la figura anterior, en el momento de intentar procesar la letra **e** como un número, se produce un error en tiempo de ejecución y se da por terminado el flujo del programa.

Veamos ahora qué sucede si modificamos el código anterior y no realizamos la conversión a entero en alguna de las variables que queremos cargar:

```
variable1=input(“Ingrese el primer valor:”)
variable2=int(input(“Ingrese el segundo valor:”))
suma= variable1 + variable2
print(“la suma de la variable 1=” , variable1, “ y la vari-
able 2=”, variable2, “es igual a “, suma)
```

Debemos notar que hemos quitado el `int(...)` al ingresar los datos de la primera variable. Veamos qué ocurre en ejecución:

```
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
celeste@cgging:~$ gedit sumaVitaminas.py
celeste@cgging:~$ python3 sumaVitaminas.py
Ingrese el primer valor:3
Ingrese el segundo valor:4
la suma de la variable 1= 3  y la variable 2= 4 es igual a  7
celeste@cgging:~$ gedit sumaVitaminas.py
celeste@cgging:~$ python3 sumaVitaminas.py
Ingrese el primer valor:3
Ingrese el segundo valor:4
Traceback (most recent call last):
  File "sumaVitaminas.py", line 3, in <module>
    suma= variable1 + variable2
TypeError: must be str, not int
celeste@cgging:~$
```

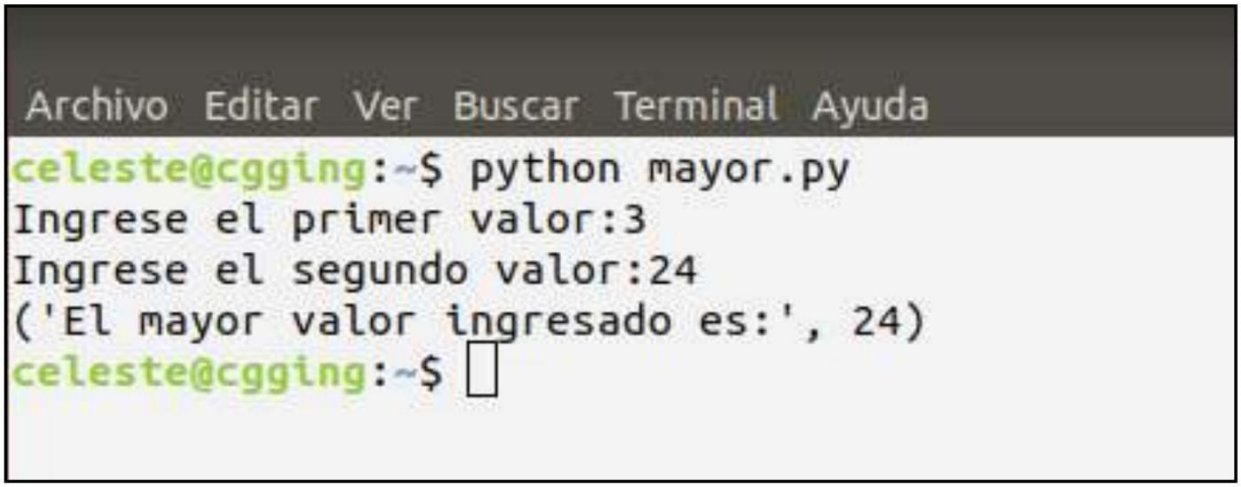
A pesar de haber ingresado dos datos numéricos, el intérprete no logra procesar el primero ya que omitimos realizar la conversión a entero y, por ende, no interpreta al valor entero **3**, sino su representación en ASCII, por lo que entiende que es un carácter y no realmente el valor **3**. Por este motivo no logra resolver la suma y genera un error en tiempo de ejecución.

OBTENER EL MAYOR

Ahora supongamos que en vez de sumar los dos valores ingresados por teclado, queremos simplemente quedarnos con el mayor de los dos. El código para dicha tarea sería:

```
variable1=int(input("Ingrese el primer valor:"))  #Se pide
el ingreso del primer valor
variable2=int(input("Ingrese el segundo valor:")) #Se pide
el ingreso del 2do valor
```

```
if (variable1>variable2):
    mayor=variable1                    #indentacion nece-
saria para marcar el bloque if
else:
    mayor=variable2                    #indentacion nece-
saria para marcar el bloque else
print("El mayor valor ingresado es:",mayor)
```



PERTENECE AL RANGO

Supongamos que tenemos un valor y queremos saber si está en un cierto rango para luego realizar alguna tarea:

```
numero=5
if(numero>0 and numero<10):     #Es posible unir condicio-
nes con operadores logicos
    print("El numero pertenece al rango")
else:
    print("El numero no pertenece al rango")
```

En el código anterior solo chequeamos si el valor pertenecía a un rango y mostramos por pantalla el mensaje que indicaba si se cumplía o no la condición.

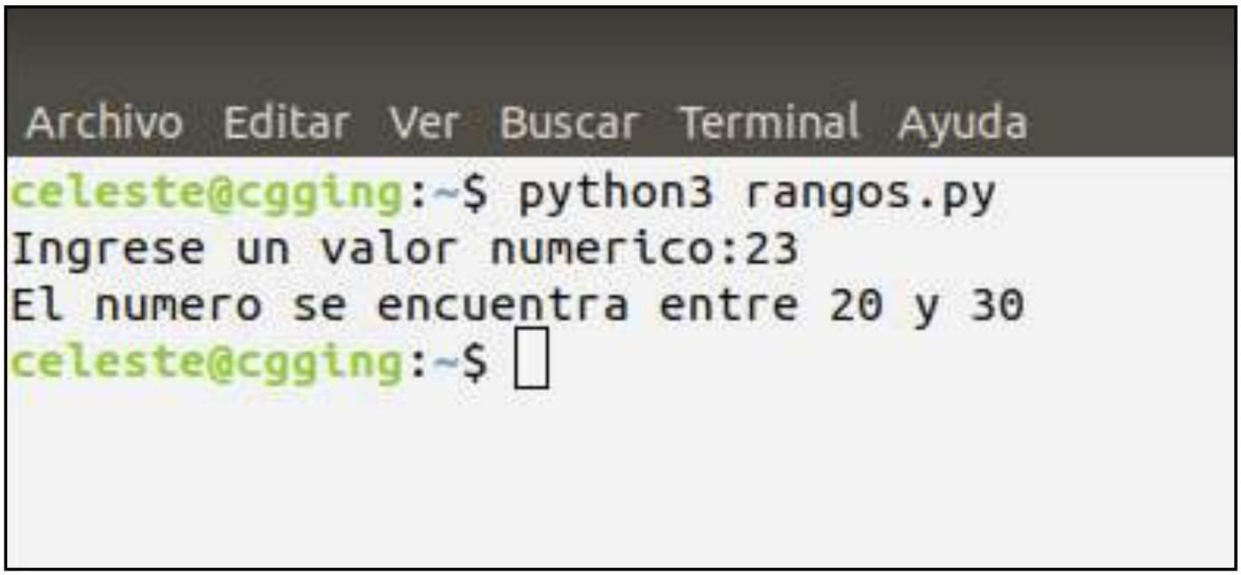
Ahora podríamos modificar el código anterior para chequear si el valor se encuentra en alguno de tres rangos predeterminados y, en caso de que así sea, que se nos muestre por pantalla a qué rango pertenece:

```
numero=5
if(numero>0 and numero<10):     #Es posible unir condicio-
nes con operadores logicos
    print("El numero se encuentra entre 0 y 10")
elif(numero>10 and numero<20):
    print("El numero se encuentra entre 10 y 20")
elif(numero>20 and numero<30):
    print("El numero se encuentra entre 20 y 30")
else:
    print("El numero no pertenece al rango")
```

En los ejemplos anteriores, el valor fue establecido directamente desde el propio código, podemos modificar el código anterior para pedir al usuario el ingreso del número:

```
numero=int(input("Ingrese un valor numerico:"))
if(numero>0 and numero<10):     #Es posible unir condicio-
nes con operadores logicos
    print("El numero se encuentra entre 0 y 10")
elif(numero>10 and numero<20):
    print("El numero se encuentra entre 10 y 20")
elif(numero>20 and numero<30):
    print("El numero se encuentra entre 20 y 30")
else:
    print("El numero no pertenece al rango")
```

Al ejecutar esta última versión de nuestro código:



Por último y para finalizar con este ejercicio de ejemplo, podemos agregar un bucle para chequear más de un valor cada vez que ejecutamos el programa:

```
i=3
while(i>0):
    numero=int(input("Ingrese un valor numerico:"))
    if(numero>0 and numero<10):
        print("El numero se encuentra entre 0 y 10")
    elif(numero>10 and numero<20):
        print("El numero se encuentra entre 10 y 20")
    elif(numero>20 and numero<30):
        print("El numero se encuentra entre 20 y 30")
    else:
        print("El numero no pertenece al rango")
    i=i-1
```

El código anterior nos pedirá el ingreso de un valor, nos dirá en qué rango se encuentra, luego nos pedirá que ingresemos otro valor, y así se repetirá el proceso tres veces.

```
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
celeste@cgging:~$ python3 rangoVarios.py
Ingrese un valor numerico:3
El numero se encuentra entre 0 y 10
Ingrese un valor numerico:34
El numero no pertenece al rango
Ingrese un valor numerico:15
El numero se encuentra entre 10 y 20
celeste@cgging:~$
```

MOSTRAR LOS NÚMEROS DEL 1 AL 100: WHILE VS. FOR

En el ejemplo anterior, utilizamos un bucle **while** para repetir la ejecución del programa tres veces, ahora queremos mostrar por pantalla los valores desde el **1** hasta el **100**:

Utilizando **while**:

```
i = 1
while( i<=100 ):
    print(i)
    i+=1
print("Fin del bucle")
```

Utilizando **for**:

```
for i in range(1,101):
    print(i)
```

La primera diferencia que podemos notar en este código en particular es que utilizando **for** podemos resolver el código en una cantidad de líneas menor que utilizando **while**. No obstante, ambas opciones son válidas para resolver este ejercicio.

MOSTRAR LOS NÚMEROS PARES ENTRE 1 Y 100

```
#1º forma
print("1 forma")
for i in range(1,101):
    if( (i%2)==0 ):
        print(i)

print("")

#2º forma
print("2 forma")
for i in range(2,101,2):
    print(i)
```

JUGAR CON RANGOS

Comenzaremos generando un rango de diez números entre **0** y **10**:

```
rango = list( range(10) )
print(rango)
```

Ahora acotaremos el rango a cinco valores entre **5** y **10**:

```
rango = list(range(5,10))
print(rango)
```

También podemos generar un rango con números decrecientes:

```
rango = list(range(10,0,-1))
print(rango)
```

O generar dos rangos separados y luego concatenarlos:

```
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
>>> rango=list(range(10,0,-1))
>>> print(rango)
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
>>> rango1=list(range(0,11))
>>> rango2=list(range(15,21))
>>> final=rango1+rango2
>>> print(final)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 16, 17, 18, 19, 20]
>>> exit()
celeste@cgging:~$ gedit range.py
celeste@cgging:~$ pyton3 range.py

Orden «pyton3» no encontrada. Quizá quiso decir:

  la orden «python3» del paquete deb «python3-minimal»
  la orden «pytone» del paquete deb «pytone»

Pruebe con: sudo apt install <nombre del paquete deb>

celeste@cgging:~$ python3 range.py
Como te llamas?   celeste
[0, 1, 2, 3, 4, 5, 6]
celeste@cgging:~$ clear

celeste@cgging:~$ python3 range.py
Como te llamas?   Celeste
[0, 1, 2, 3, 4, 5, 6]
celeste@cgging:~$
```

```
rango1 = list(range(0,11))
rango2 = list(range(15,21))
final = rango1 + rango2
print(final)
```

Lo que vamos a hacer a continuación es solicitarle el nombre al usuario y luego generar un rango desde **0** hasta la longitud de su nombre:

```
nombre=input("Como te llamas?   ")
rango = list( range(0, len(nombre)))
print(rango)
```

CADENAS DE CARACTERES

Vamos a comenzar pidiendo al usuario dos cadenas de caracteres por teclado, luego intercambiaremos los primeros caracteres en ambas cadenas y las mostraremos concatenadas por pantalla con un espacio entre ambas:

```
cadena1 = input("Dame la primera cadena: ")
cadena2 = input("Dame la segunda cadena: ")
print( cadena2[:2] + cadena1[2:] + " " + cadena1[:2] + cade-
na2[2:] )
```

Para seguir, pediremos al usuario que ingrese una cadena de caracteres por teclado e le informaremos si la cadena que ingresó es un palíndromo:

```
cadena1 = input("Dame una cadena: ")
cadena_al_reves = cadena1[::-1]
print(cadena_al_reves)
if( cadena1 == cadena_al_reves ):
    print("Es palindromo")
else:
    print("No es palindromo")
```

EJEMPLO FINAL: INTEGRACIÓN DE TODO LO VISTO Y ALGO MÁS

El código que presentaremos a continuación genera aleatoriamente un valor y espera que el usuario sea capaz de adivinarlo. En cada intento erróneo se le indica al usuario si el valor que está buscando es mayor o menos al que ingresó y, una vez acertado, se le informa la cantidad de intentos que requirió para adivinar el número:

```
from random import *
def generaNumeroAleatorio(minimo, maximo):
return randint(minimo, maximo)

numero_buscado=generaNumeroAleatorio(1,100)
encontrado=False
intentos=0

while not encontrado:
```

```
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
celeste@cgging:~$ python ejemplo.py
Introduce el numero buscado:  100
El numero que buscas es menor
Introduce el numero buscado:  40
El numero que buscas es menor
Introduce el numero buscado:  20
El numero que buscas es menor
Introduce el numero buscado:  15
El numero que buscas es menor
Introduce el numero buscado:  10
El numero que buscas es menor
Introduce el numero buscado:  4
El numero que buscas es menor
Introduce el numero buscado:  2
('Has acertado, el numero correcto es ', 2, 'te ha llevado', 6, 'intentos')
celeste@cgging:~$
```

```
    numero_usuario=int(input("Introduce el numero buscado:  "))
    if numero_usuario>numero_buscado:
        print("El numero que buscas es menor")
        intentos=intentos+1
    elif numero_usuario<numero_buscado:
        print("El numero que buscas es mayor")
        intentos=intentos+1
    else:
        encontrado=True
        print("Has acertado, el numero correcto es ", numero_
usuario, "te ha llevado", intentos, "intentos")
```

En este ejemplo sumamos varias cosas interesantes: para empezar importamos un módulo, además sumamos el uso de funciones.

Comencemos analizando el porqué del modulo. Dijimos que este código debe generar un número aleatorio, para eso Python posee un módulo llamado **random** que cuenta con diferentes funciones para crear y manejar números aleatorios.

Para poder utilizar los elementos de un módulo en un código que queramos realizar, en primer lugar debemos importar el módulo. Por ese motivo para poder utilizar la función `randint()` en el código anterior, necesariamente necesitamos importar el módulo random.

Lo siguiente que vemos en este código es la definición de una función. Si bien el tema funciones escapa a los alcances de este libro, mencionaremos lo básico sobre cómo construir una función.

Una **función** es la forma de agrupar expresiones y sentencias (algoritmos) que realicen determinadas acciones, pero que estas solo se ejecuten cuando son llamadas. Es decir que, al colocar un algoritmo dentro de una función y al correr el archivo, el algoritmo no será ejecutado si no se ha hecho una referencia a la función que lo contiene.

En Python, la definición de funciones se realiza mediante la instrucción `def` más un nombre de función descriptivo –para el cual se aplican las mismas reglas que para el nombre de las variables– seguido de paréntesis de apertura y cierre. Como toda estructura de control en Python, la definición de la función finaliza con dos puntos (`:`) y el algoritmo que la compone irá indentado con cuatro espacios:

```
def mi_funcion():
    # aquí el algoritmo
```

Una función no es ejecutada hasta tanto no sea invocada. Para invocar una función, simplemente se la llama por su nombre:

```
def mi_funcion():
    print “Hola Mundo”

funcion()
```

Cuando una función haga un retorno de datos, estos pueden ser asignados a una variable:

```
def funcion():
    return “Hola Mundo”

frase = funcion()
print frase
```

Una función puede tener cualquier tipo de algoritmo y cualquier cantidad de ellos, y utilizar alguna de las características vistas hasta ahora. No obstante, una buena práctica indica que la finalidad de una función debe ser realizar una única acción, reutilizable y, por lo tanto, tan genérica como sea posible.

RedUSERS PREMIUM

- Cientos de publicaciones USERS por una mínima cuota mensual.
- Siempre, donde vayas. On Line - Off Line. En cualquier dispositivo.
- Al menos 1 novedad semanal. Son 680 publicaciones y sumando...!!
- Incluye: eBooks - Informes USERS - Guías USERS - Revistas USERS y Power – CURSOS

SUSCRÍBETE

usershop.redusers.com

+54-11-4110-8700

usershop@redusers.com

USERS

VOL. I

Programación en python

ACERCA DE ESTE CURSO

Python es un lenguaje multiplataforma y multiparadigma sumamente versátil que se ha vuelto muy popular en los últimos tiempos, debido, entre otros motivos, a que se convirtió en el lenguaje de elección para las aplicaciones de Inteligencia Artificial.

Este curso de Python en tres volúmenes permite aprender desde cero lo necesario para aprovechar el potencial de este lenguaje de programación, contando con ejemplos prácticos que nos ayudarán a comprender, desde las propias bases del lenguaje, hasta temas avanzados como el paradigma de programación orientada a objetos.

EL VOLUMEN I

En este primer volumen de los tres que componen la serie, exploramos los conceptos básicos de la sintaxis, necesarios para enfrentarnos al mundo del desarrollo de scripts, analizamos las características más importantes de Python y nos familiarizamos tanto con el intérprete interactivo como con el desarrollo y ejecución de scripts. También revisamos la sintaxis, elementos y estructuras de control de este lenguaje.

SOBRE LA AUTORA

Celeste Guagliano es Ingeniera, docente y entusiasta de la tecnología. Se desempeña como profesora universitaria enseñando lenguajes de programación variados tales como Assembler, C, Java y Python. También ha trabajado como docente de posgrado en el curso de Técnicas de Programación Científica junto la dra Cecilia Jarne, su amiga y colaboradora en el primer volumen de esta serie.

ISBN 978-987-4958-09-9