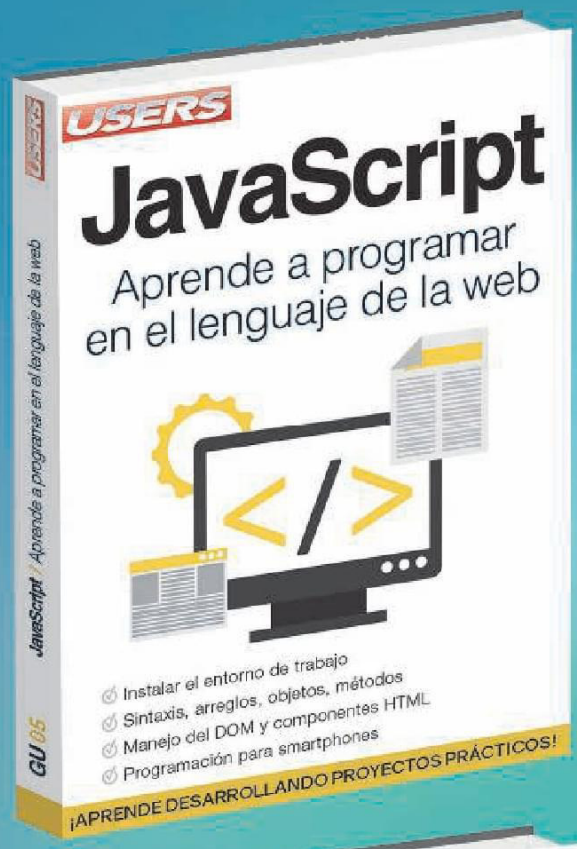
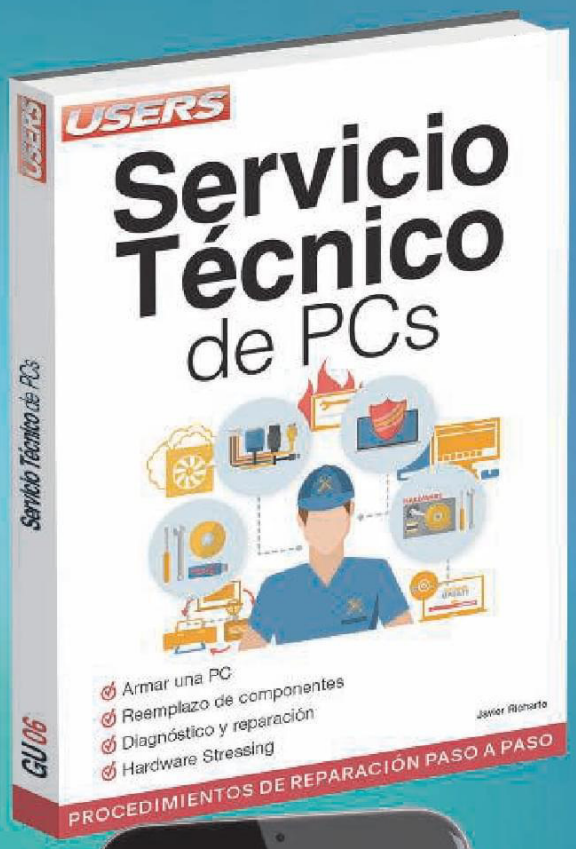
USERS

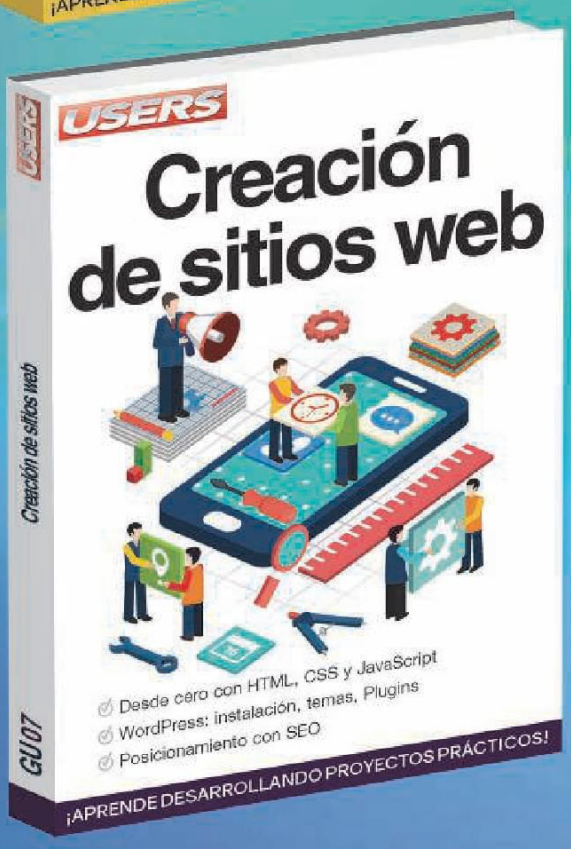
VOL. II

Programación en python

PROGRAMACIÓN ORIENTADA A OBJETOS







SUSCRIBETE

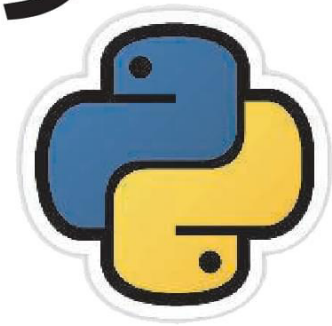
usershop.redusers.com

+54-11-4110-8700

usershop@redusers.com

CURSO DE PROGRAMACION PYTHON

Programación en python



PROGRAMACIÓN ORIENTADA A OBJETOS

Título: Programación en Python - Vol. II / **Autor**: Celeste Guagliano
Coordinador editorial: Miguel Lederkremer / **Edición**: Claudio Peña
Diseño y Maquetado: Marina Mozzetti / **Colección**: USERS ebooks - LPCU294

Copyright © MMXIX. Es una publicación de Six Ediciones. Hecho el depósito que marca la ley 11723. Todos los derechos reservados. Esta publicación no puede ser reproducida ni en todo ni en parte, por ningún medio actual o futuro, sin el permiso previo y por escrito de Six Ediciones. Su infracción está penada por las leyes 11723 y 25446. La editorial no asume responsabilidad alguna por cualquier consecuencia derivada de la fabricación, funcionamiento y/o utilización de los servicios y productos que se describen y/o analizan. Todas las marcas mencionadas en este libro son propiedad exclusiva de sus respectivos dueños. Libro de edición argentina.

Guagliano, Celeste
Programación en Python 2 : programación orientada a objetos / Celeste Guagliano. -
1a ed . - Ciudad Autónoma de Buenos Aires :
Six Ediciones, 2019.
Libro digital, PDF - (Programación en Python ; 2)

Archivo Digital: online
ISBN **978-987-4958-19-8**

1. Informática. 2. Lenguajes de Programación.
I. Título.
CDD 005.4

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, 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 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.

¡Éxitos en este nuevo desafio!

Sumario

Vol. II

INTRODUCCIÒN A LA POO / 8

¿QUÉ ES LA PROGRAMACIÓN ORIENTADA A OBJETOS? / 8

MODELAR LA REALIDAD / 9

UN POCO DE HISTORIA SOBRE LA POO / 10

PROFUNDIZAR EL PARADIGMA POO / 11

POO EN PYTHON / 13

ALGUNAS PARTICULARIDADES DE POO EN PYTHON

OBJETOS, CLASES Y FUNCIONES / 16

OBJETOS / 16

CLASES / 17

HERENCIA Y CREACIÓN DE CLASES

MÉTODOS / 20

CARGAR CÓDIGO EN LA CONSOLA A TRAVÉS DE UN ARCHIVO

ÁMBITO DE LOS MÉTODOS / 21

MÉTODOS ESPECIALES

VARIABLES DE CLASE Y DE INSTANCIA / 23

DATOS PARA TENER EN CUENTA

HERENCIA / 26

¿QUÉ ES LA HERENCIA Y PARA QUÉ SIRVE? / 26

HERENCIA MÚLTIPLE / 29

VARIABLES PRIVADAS / 30

ITERADORES / 31

GENERADORES / 33

EXPRESIONES GENERADORAS / 34

MANEJO DE ARCHIVOS / 36

COMPRENDER ARCHIVOS / 36

MÓDULOS Y BIBLIOTECA ESTÁNDAR / 40

MÓDULOS / 40

MÁS SOBRE LOS MÓDULOS / EJECUTAR MÓDULOS COMO SCRIPTS / EL CAMINO DE BÚSQUEDA DE LOS MÓDULOS / MÓDULOS ESTÁNDARES / INTERFAZ AL SISTEMA OPERATIVO / COMODINES DE ARCHIVOS / ARGUMENTOS DE LÍNEA DE ÓRDENES / REDIRECCIÓN DE LA SALIDA DE ERROR Y FINALIZACIÓN DEL PROGRAMA / COINCIDENCIA EN PATRONES DE CADENAS / MATEMÁTICA / ACCESO A INTERNET / FECHAS Y TIEMPOS / COMPRESIÓN DE DATOS / MEDICIÓN DE RENDIMIENTO

CONTROL DE CALIDAD / 52

LAS PILAS INCLUIDAS / 53

ERRORES COMUNES Y BUENAS PRÁCTICAS / 54

ROMPER CÓDIGOS / 54

ERRORES AL INTERNTAR EJECUTAR UN SCRIPT DE PYTHON / HOLA MUNDO "CRUEL" / SUMA

EXCEPCIONES / 59

MANEJAR EXCEPCIONES / OBTENER EL MAYOR / EJEMPLO FINAL: SUMAR LA DOCUMENTACIÓN DEL CÓDIGO

TESTEAR CÓDIGOS / 72

PRUEBAS AUTOMÁTICAS VERSUS

PRUEBAS MANUALES / 72

PRUEBAS UNITARIAS VERSUS PRUEBAS DE INTEGRACIÓN / 73

PROFUNDIZAR EL CONCEPTO DE UNIT TESTING / 74

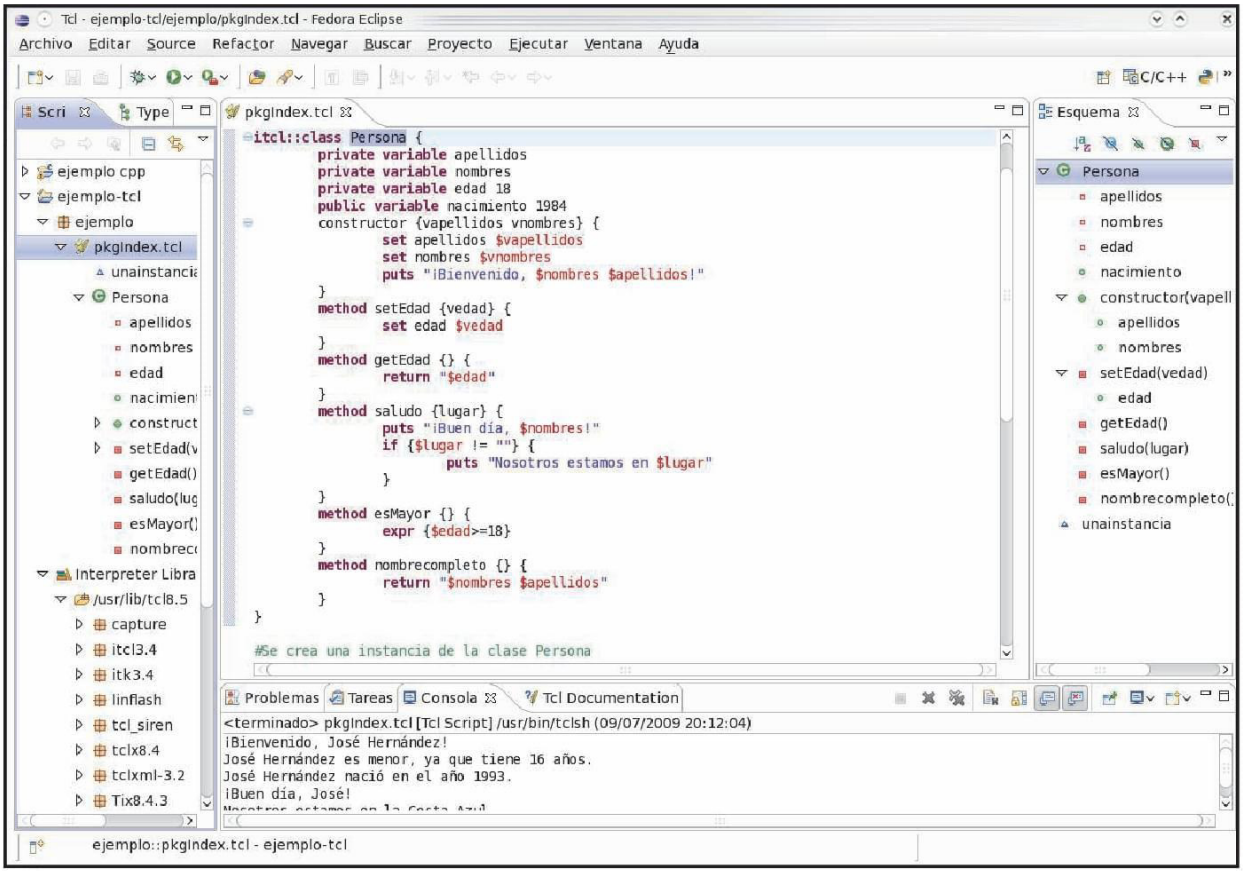
UTILIZAR UNIT TESTING / IMPLEMENTAR UNIT TESTING EN PYTHON / PRUEBA UNITARIA CON UNITTEST

PRUEBA UNITARIA CON DOCTEST / 81

Introducción a la POO

¿QUÉ ES LA PROGRAMACIÓN ORIENTADA A OBJETOS?

Hasta el momento hemos trabajado con el paradigma de programación estructurada o procedural. Este resulta de mucha utilidad en ámbitos muy diversos, aunque en algunas cuestiones puede ser complejo de aplicar o, incluso, complicarnos la tarea que queremos realizar.



01

En el volumen I hicimos una recorrida por los conceptos básicos y fundamentales para dar nuestros primeros pasos en Python. Este volumen pretende recorrer Python en profundidad y, para eso, nada mejor que comenzar a estudiar el paradigma de programación orientada a objetos.

Supongamos que estamos trabajando en un proyecto extenso y que, a causa del tiempo con el que contamos y de la magnitud del proyecto, necesitamos formar un equipo de trabajo. Supongamos además que ese equipo de trabajo se encuentra repartido en distintos lugares y que no tenemos comunicación constante entre los integrantes ni la facilidad de ponernos de acuerdo con una determinada tarea como si estuviéramos trabajando en el mismo horario y lugar físico. En el escenario planteado, deberíamos dividir la tarea de programación, y luego unir todo y hacerlo funcionar. Si bien no es una tarea imposible, el paradigma estructurado no está optimizado para este método de trabajo, por lo que puede resultar bastante engorroso ensamblar las partes y que todo funcione según lo esperado.

Con esta problemática en mente, surgió otro paradigma que conocemos como **paradigma de programación orientada a objetos (POO)**.

Este paradigma está basado en varias técnicas tales como: herencia, cohesión, abstracción, polimorfismo, acoplamiento y encapsulación.

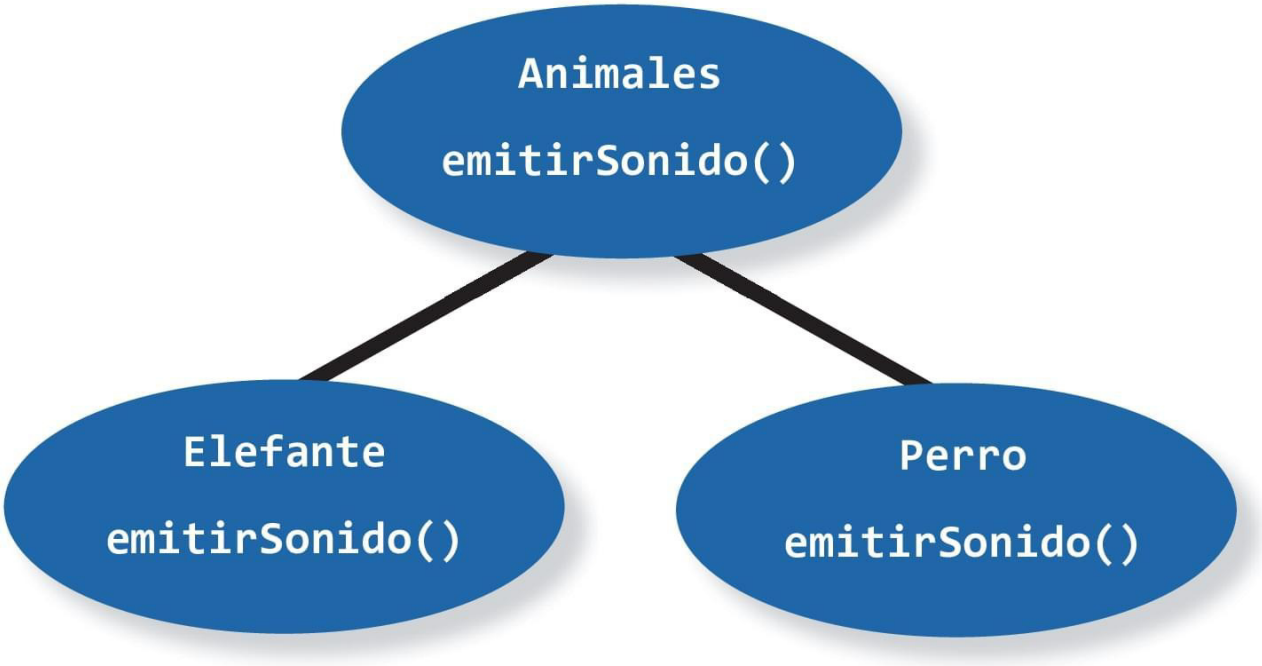
La motivación para implementar el paradigma POO tiene dos principales fundamentos: organizar el código fuente y reusar código fuente en similares contextos.

MODELAR LA REALIDAD

El paradigma de programación orientado a objetos consiste en otra forma de planear un algoritmo. En el paradigma estructurado, nuestros algoritmos se concebían como una serie de pasos secuenciales, que se ejecutarían uno tras otro. En POO se piensa en un modelo del problema en el cual tendremos ciertos objetos que representarán diferentes partes de la solución al problema que se nos plantea, y dichos **objetos** son capaces de comunicarse entre sí mediante **mensajes**.

A su vez, los objetos manipulan los datos de entrada para la obtención de datos de salida específicos, donde cada objeto ofrece una funcionalidad especial.

El modelo que utilizaremos para resolver nuestro problema en este paradigma es mucho más cercano a la realidad de lo que nos podría resultar en cualquier otro. Así, si nuestro problema consiste en realizar un estudio sobre aves, nuestro objeto será, por ejemplo, "golondrina" y podremos preguntarle cuánto comió o cuánto voló. Este modelo de la golondrina real sabrá indicarnos estos parámetros.



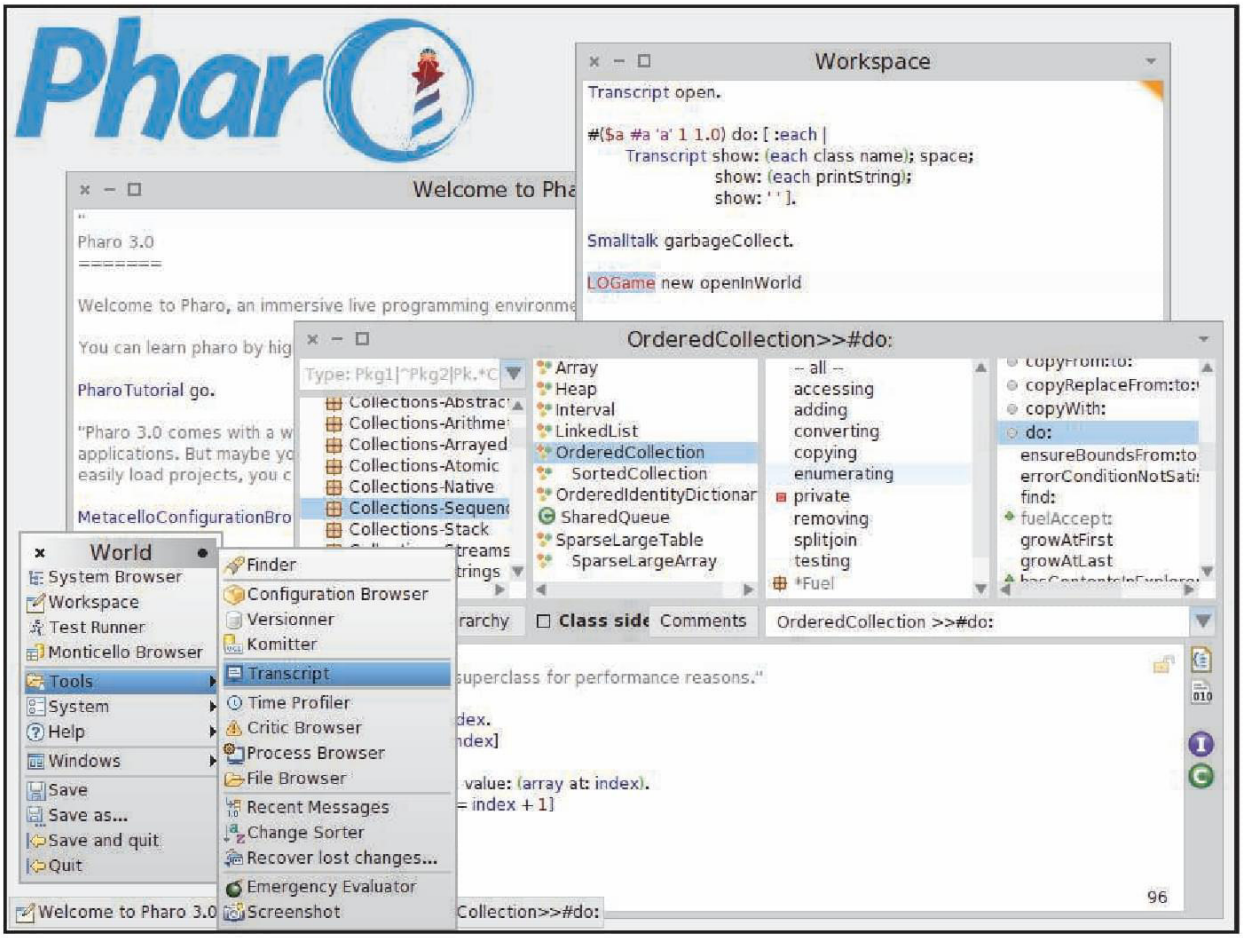
UN POCO DE HISTORIA SOBRE POO

Uno de los lenguajes precursores en programación orientada a objetos, fue **Smalltalk**. Como mencionamos antes, el POO se basa en objetos y mensajes; Smalltalk significa en inglés 'pequeña charla' y se refiere justamente a estos objetos que se envían mensajes entre sí para llevar adelante la resolución del problema que modelamos.

Con Smalltalk se desarrolló gran parte de la teoría de objetos entre los años 1972 y 1980, pero el uso de este paradigma recién se popularizó a principios de los 90.

Logotipo del popular lenguaje C++ basado en objetos.

Logotipo del popular lenguaje Java basado en objetos.



En la actualidad podemos encontrar una gran variedad de lenguajes de programación en los cuales es posible utilizar el paradigma de programación orientada a objetos. Entre ellos, cabe mencionar **C++**, **Java** y, claro está, el lenguaje que nos motivó a escribir esta obra: **Python**.

PROFUNDIZAR EN EL PARADIGMA POO

Ya nombramos los objetos y los mensajes, pero, si bien estos representan la base del paradigma, es natural pensar que el paradigma es más complejo y cuenta con mayor cantidad de elementos.

Podemos nombrar algunos de estos elementos que nos resultarán muy familiares en poco tiempo. Ellos son: **clases**, **instancias**, **atributos** y **métodos**. Claro está que vamos a toparnos con más elementos en este recorrido, pero para comenzar podemos quedarnos con estos cuatro.

```
1   sub Calculator()
2       sub addition(self, other)
3           return self + other
4       end
5
6       /** Create a list of 2 numbers. */
7       sub makeFraction(numerator, denominator)
8           return [numerator, denominator]
9       end
10
11      /** Warning: Destroys original fraction! */
12      sub multiplyFracs(frac, otherFrac)
13          frac[0] *= otherFrac[0]
14          frac[1] *= otherFrac[1]
15          return frac
16      end
17  end
18
19  sub InfinityCalculator()
20      inherit Calculator()
21      /** Create a list of 2 numbers. */
22      sub makeFraction(numerator, denominator)
23          if denominator == 0
24              /* The user is trying to divide by 0.
25               * Use Java's way of handling this: */
26              import math into mathematics
27              return mathematics.INFINITY
28          end
29          return [numerator, denominator]
30      end
31  end
```


Vamos a ver que trabajar con objetos tiene cierta similitud a trabajar con funciones, aunque tenemos que aprender a nombrar cada cosa y sus características de uso.

A simple vista notamos que, en estos cuatro elementos que mencionamos, no figuran los objetos… ¿o sí? En realidad sí: los objetos son lo que mencionamos como *instancias* en el grupo de los cuatro fantásticos. Pero entonces, ¿de qué es instancia un objeto? Es posible explicarlo de esta manera, nosotros vamos

a manipular objetos que probablemente no sean únicos, pero que necesitamos individualizarlos. Por ejemplo, unos párrafos más arriba hablamos sobre el objeto `Golondrina`, que sabrá volar, comer e informarnos cuánto comió o voló en el día. Pero golondrinas hay muchas y, como podemos suponer, todas sabrán comer y volar.

Una de las características principales del paradigma POO tiene que ver con la reutilización de código, entonces, si tenemos una bandada de golondrinas, ¿debemos programar el comportamiento de cada una? La respuesta es: "¡Claro que no!". Para eso tenemos las clases.

Una clase es un molde que nos sirve para instanciar un objeto, entonces si tenemos una bandada de golondrinas, es natural pensar que la clase en cuestión será `Golondrina`, y los objetos con los que trabajaremos serán las instancias de esta clase, a las cuales bautizaremos con nombres diferentes y arbitrarios (`gaviota1`, `pepita`, `larry`...). Todas serán individuos independientes y, si bien cada una sabrá realizar las mismas acciones y entenderá los mismos mensajes, también será responsabilidad de ellas conocer su propio estado y el de nadie más.

Los atributos por su parte serían equivalentes a las variables que utilizábamos en las funciones. Son datos que necesitamos almacenar ya que no pueden calcularse, sino que tienen que ser proporcionados. Por ejemplo, en el caso de la golondrina, uno de sus atributos podría ser **energía**.

Por último, falta mencionar a los métodos. Estos describen el comportamiento de los objetos de una clase: explicitan las operaciones que los objetos de una clase son capaces de realizar.

Hasta aquí hemos tenido un primer acercamiento a la comprensión del paradigma orientado a objetos. Veamos ahora cómo aplicar POO en Python.

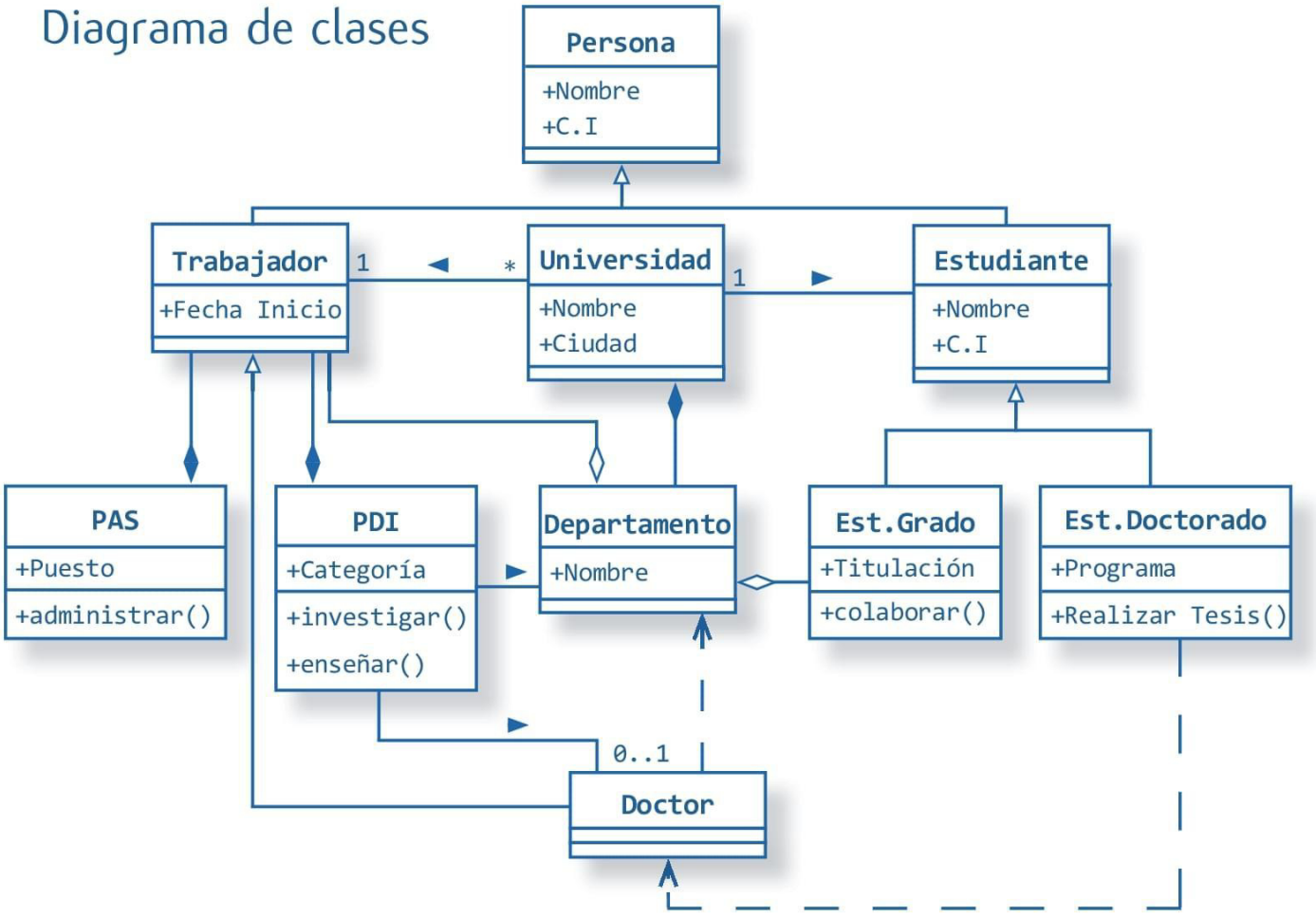
POO EN PYTHON

El mecanismo de clases de Python agrega clases al lenguaje con un mínimo de nuevas sintaxis y semánticas. Si ya hemos programado en el paradigma POO, encontraremos que en Python las clases resultan una mezcla de los mecanismos encontrados en otros lenguajes. Como es cierto para los módulos, las clases en Python no ponen una barrera absoluta entre la definición y el usuario, sino que más bien se apoya en la cortesía del usuario de no forzar la definición. Sin embargo, se mantiene el poder completo de las características más importantes de las clases: el mecanismo de la herencia de clases permite múltiples clases base;

una clase derivada puede sobrescribir cualquier método de sus clases base; y un método puede llamar al método de la clase base con el mismo nombre.

Lo anterior parece complejo ya que vimos mucha información nueva y toda junta. Vayamos por partes para comprender mejor.

Diagrama de clases



¿Qué significa *herencia*? Si lo pensamos en términos cotidianos, es normal escuchar que un hijo heredó los ojos de la madre o las orejas del padre, la sonrisa del abuelo, etcétera. En este caso se refiere a esto mismo, pero entre clases. Y ¿por qué podemos hablar de que haya clases que heredan de otras? La idea del paradigma es reutilizar todo el código posible y reducir cuanto se pueda cualquier esfuerzo inútil. Con esto en mente, retomemos el ejemplo de la golondrina; supongamos que ahora, además de golondrinas, tenemos canarios, que comen y vuelan, pero también saben cantar. Podemos pensar la clase Canario como una clase derivada o "hija" de golondrina. Es posible hacer esto porque el canario sabe las mismas cosas que la golondrina y, además, se le agrega un comportamiento particular, como cantar.

La herencia nos permite modelar otros sistemas sin tener que empezar de cero, utilizamos una clase con las características de base y le agregamos las características particulares a la **clase hija**.

Ahora bien, si leímos con atención, habremos notado que hablamos de **clases base** en plural, o sea, que una clase puede tener múltiples **clases padre**. Para entender esto, comparemos con el ejemplo de uso cotidiano, cuando hablamos de herencia por parte de madre, padre, abuelo, etcétera.

Para llevarlo a un modelo de objetos más concreto, pensemos en un auto anfibio. Los anfibios son aquellos vehículos capaces de movilizarse por tierra o por agua. Entonces, al hablar de auto, resulta inmediato plantear la clase `Auto`, pero, si tenemos en cuenta que puede desplazarse en el agua, podríamos también pensar en la clase `Lancha`. Aquí tenemos el ejemplo de herencia múltiple, ya que en este caso el anfibio se comporta como ambos, es decir que sus clases base serán tanto `Lancha` como `Auto`.

Pero, además, también es posible que la clase hija tenga un cierto comportamiento diferente de la clase de base o padre. Volviendo al ejemplo de los pájaros, tanto la golondrina como el canario comen, pero tal vez el canario y la golondrina coman diferentes cosas o de diferente forma, ese comportamiento se diferencia desde el método y, para eso, desde la clase hija podremos redefinir ese método. Es decir que podemos hacer que los canarios (cualquier objeto de la clase `Canario` o cualquier instancia de `Canario`) coman de forma diferente que las golondrinas.

```
class Golondrina:
pass
```

ALGUNAS PARTICULARIDADES DE POO EN PYTHON

- **Todo es un objeto, incluyendo los tipos y las clases.**
- **Permite herencia múltiple.**
- **No existen métodos ni atributos privados.**
- **Los atributos pueden ser modificados directamente.**
- **Permite *monkey patching*.**
- **Permite *duck typing*.**
- **Permite la sobrecarga de operadores.**
- **Permite la creación de nuevos tipos de datos.**

Objetos, clases y funciones

OBJETOS

Los objetos son el corazón del paradigma, pero eso no parece ser ninguna novedad considerando que hablamos justamente del paradigma orientado a objetos.

Si lo pensamos como algo cotidiano, un objeto es, en nuestro entorno, todo aquello que cumple una función y tiene características propias. Así podemos comprender que todo es un objeto: una mesa, un auto, un celular. Incluso, y aunque suene extraño o incorrecto con la definición antes dada, podemos considerar que nuestras mascotas y las personas son objetos.

En el paradigma, modelamos nuestros problemas con objetos y, de esta forma, los resolvemos de una manera más cercana a como lo haríamos en la realidad, se podría decir que es una forma más natural de programar.

Sin embargo, los objetos que crearemos no serán recreaciones idénticas de la realidad, sino que solo les daremos aquellas características que nos sirvan para la resolución de nuestro problema puntual.

Por ejemplo, si el objeto por modelar es un auto y el problema que resuelve es el traslado, no nos interesará atribuirle la característica de color, pero sí nos resultará útil saber cuánto combustible tiene.

Ya hemos trabajado utilizando objetos con anterioridad, pero no éramos conscientes de ello. Nuestros primeros pasos con Python en el e-book anterior se desarrollaron utilizando números y operaciones. Esos números y todos los tipos de datos que hemos utilizado no son otra cosa que objetos.

Hay tres elementos fundamentales que definen a un objeto: una **identidad**, un **tipo** y un **valor**.

La identidad del objeto no cambia mientras esté vivo en el programa que lo involucre. El tipo de un objeto es inmutable y determina las operaciones que admite (por ejemplo, "¿tiene una longitud?"), además define los valores posibles para los objetos de ese tipo. La función **type()** devuelve el tipo de un objeto (que es un objeto en sí mismo).

En el capítulo anterior dejamos sentadas las bases del paradigma de programación orientada a objetos (POO). En este capítulo veremos en profundidad los conceptos fundamentales aplicados a Python y daremos nuestros primeros pasos con clases y objetos.

02

Algunos objetos pueden cambiar de valor, es decir que son mutables; aquellos que no tienen esta característica se consideran inmutables.

Hasta acá podríamos decir que sabemos trabajar con objetos porque ya lo hemos hecho con anterioridad, aunque hemos trabajado con objetos preconstruidos. ¿Qué sucede si queremos generar nuestro propio objeto?, y yendo aún más lejos, ¿qué sucede si tenemos varios objetos que se comportan de la misma manera? ¿Debemos programar uno por uno?

Para responder todo eso y comenzar a trabajar en el código, es necesario comprender el concepto de clases.

CLASES

Las **clases** son los "moldes" de los objetos, es decir, una clase definirá todo aquello que un objeto debe ser, conocer y realizar, pero no podrá ejecutarlo por sí misma. Para lograr un objeto, es necesario instanciar una clase.

Y nos preguntaremos, ¿pero por qué es necesario instanciar la clase?, ¿no es verdad acaso que todo es un objeto?

Sí, es cierto, pero la clase objeto tiene otro tipo de características que nos servirán para reutilizar y organizar nuestro código.

Veamos un ejemplo, definamos una clase para comprender un poco más de qué estamos hablando:

Para definir una clase en Python, utilizamos la palabra reservada **class**, y el resto se asemeja mucho a la declaración de una función.

```
class Persona:
    pass
```

Analicemos lo anterior: `Persona` será el nombre de la clase y, como podemos ver, aún no tiene nada programado, sino que pusimos la sentencia `pass` para indicar que todavía no hace nada, pero existirá dicha clase en nuestro programa.

Un dato para tener en cuenta es que, por convención y buenas prácticas, los nombres de las clases se escriben con mayúsculas, así sabremos de un vistazo si algo es una clase o una instancia.

Otra buena práctica, ya que las clases las generaremos nosotros y serán específicas de lo que queramos representar, es realizar comentarios en el código, puntualmente al comienzo de la clase para explicitar su rol.

Para definir una clase y asociarle todas las características necesarias para trabajar, tenemos que introducir dos conceptos: el **estado** de un objeto y los **atributos**.

El conjunto de datos y objetos relacionados con un objeto en un momento dado se conoce como *estado*. Comprenderemos que un objeto puede ir variando su estado a lo largo de su existencia, ya que este se verá afectado según su relación con el entorno y otros objetos.

Por su parte, los atributos son las propiedades o características que puede tener un objeto. Por ejemplo, en el caso de una persona, los atributos podrían ser: nombre, apellido, sexo, nacionalidad, etcétera.

Mediante los atributos se describe el estado de un objeto.

Sigamos ahora con la clase `Persona`:

```
class Persona:
    """Clase que representa una Persona"""
    cedula = "V-13458796"
    nombre = "Leonardo"
    apellido = "Caballero"
    sexo = "M"
```

HERENCIA Y CREACIÓN DE CLASES

Si queremos crear una clase que herede las propiedades de otra clase, que se conocerá en ese contexto como **superclase**, entonces la estructura de la definición de nuestra clase será de la siguiente manera:

```
class<Nombre>(<superclase_1>, <superclase_2>... <superclase_n>):
    ...
    ...
```

Recordemos que Python permite herencia múltiple, por ese motivo puede haber varias superclases en la definición.

En el caso que no queramos especificar ninguna superclase, simplemente omitiremos los paréntesis, y nuestra clase naciente heredará de manera implícita de la clase **Object**.

Si quisiéramos probar el código anterior desde la consola interactiva para empezar a comprender el uso de clases y objetos antes de la creación de un proyecto, desde la consola interactiva tipearíamos:

```
>>>class Persona:
...     """Clase que representa una Persona"""
...     dni = "12345678"
...     nombre = "Juan"
...     apellido = "Perez"
...     sexo = "M"
…
```

Hasta acá realizamos la definición de la clase **Persona** con la que nos interesa trabajar. Notemos que hemos cargado los datos de una persona hipotética a modo de plantilla para que los atributos de **Persona** tengan el tipo de dato necesario para cada uno. Ahora debemos crear una instancia para ver cómo podremos interactuar con los objetos de dicha clase:

```
>>>juancito = Persona()
```

¿Qué ocurrirá ahora si queremos saber el nombre o el DNI de la instancia que acabamos de crear?

Por lo pronto, no tenemos forma de acceder a dichos datos porque, si bien contamos con los atributos declarados, no hay ninguna función que los acceda o los setee. Para ello debemos introducir el concepto de método y comenzar a ampliar la definición de nuestra clase.

MÉTODOS

Los métodos describen el comportamiento de los objetos de una clase y representan las operaciones que se pueden realizar con los objetos.

La ejecución de un método puede conducir a cambiar el estado del objeto.

Se definen de la misma forma que las funciones normales, pero deben declararse dentro de la clase, y su primer argumento siempre referencia a la instancia que la llama; por eso se afirma que los métodos son funciones adjuntadas a objetos.

Si el objeto es **Persona**, los métodos pueden ser: hablar, caminar, comer, dormir, etcétera.

Comenzaremos por los métodos básicos, aquellos que nos permiten cargar o mostrar los datos primordiales de la persona:

```
>>>class Persona:
...    """Clase que representa una Persona"""
...    dni = "1345678"
...    nombre = "Juan"
...    apellido = "Perez"
...    sexo = "M"
...    def nombre(self, name):
...        """cargar el nombre de la persona"""
...        self.nombre=name
...    def apellido(self, lname):
...      """cargar apellido de la persona"""
...      self.apellido=lname
...   defmostrarNombre(self):
...      """Imprimir el nombre de la persona"""
...      print(self.nombre)
...   defmostrarApellido(self)
...      """Imprimir el apellido de la persona"""
...      print(self.apellido)
```

Como vemos, la declaración de un método es muy similar a una función; la principal y única diferencia sintáctica entre uno y otro es que el primer parámetro del método por convención es **self**.

Probemos lo anterior en la consola interactiva:

```
>>> pepe=Persona()
>>>pepe.nombre("jose")
>>>pepe.mostrarNombre()
jose
```

CARGAR CÓDIGO EN LA CONSOLA A PARTIR DE UN ARCHIVO

Para no tener que copiar toda la definición de clase a la consola interactiva cada vez que queremos probar alguna modificación en la declaración de nuestra clase, podemos crear un script como vimos en el e-book 1.

Crearemos el archivo **`persona.py`** en donde copiaremos y editaremos toda la definición de clase. Para cargar esto en la consola interactiva y poder instanciar nuestro objeto **`Persona`**, simplemente tipearemos:

```
python3 -i persona.py
```

Esto nos dejará en la consola interactiva luego de la ejecución del script y podremos continuar trabajando ya con la definición de la clase precargada.

ÁMBITOS DE LOS MÉTODOS

Los métodos cuentan con un espacio de nombres propio. En caso de no encontrar un nombre en su ámbito local, buscará en el ámbito superior hasta encontrar alguna coincidencia. Por **ámbito superior** debemos entender las superclases de las cuales hereda nuestra clase o, en el caso de no haber definido ninguna, entonces de la clase **`Object`**.

Los métodos pueden acceder y crear atributos dentro del objeto al que pertenecen, anteponiendo la palabra **`self`** y el operador de atributo antes del nombre del atributo en cuestión. Esto ya lo hemos realizado en el primer ejemplo, sin formalizar la explicación de lo que estábamos haciendo.

MÉTODOS ESPECIALES

Las clases en Python cuentan con múltiples métodos especiales, que se encuentran entre dobles guiones bajos **__<metodo>__()**.

Los métodos especiales más utilizados son: **__init__()**, **__str__()** y **__del__()**.

El método **__str__()** es un método especial, que se ejecuta en el momento en el que un objeto se manda a mostrar, es decir, es una cadena representativa de la clase, la cual puede incluir formatos personalizados de presentación.

```
def __str__(self):
        """Devuelve una cadena representativa de Persona"""
return "%s: %s, %s %s, %s." % (
self.__doc__[25:34], str(self.cedula), self.nombre,
self.apellido, self.getGenero(self.sexo))
```

El método **__del__()** es un método especial, que se ejecuta en el momento en el que un objeto es descartado por el intérprete. El comportamiento de **__del__()** es muy similar a los **destructores** en otros lenguajes.

Lo que vimos con el ejemplo que estamos realizando es cómo instanciar una clase que ya tiene unos ciertos datos predefinidos y, luego, configurar "a mano" esos datos. Pero ¿no sería mucho más limpio y rápido si pudiéramos instanciar el objeto con algún método que nos permitiera en ese paso cargar toda la información requerida? Algo como esto:

```
>>> persona1 = Persona("23156789", "Maria", "Antonieta", "F")
```

No es imposible de lograr, ya que contamos con el método **init**, que actúa a modo de constructor de la clase. El método **__init__()** es un método especial, que se ejecuta en el momento de instanciar un objeto. El comportamiento de **__init__()** es muy similar a los "constructores" en otros lenguajes. Los argumentos que se utilizan en la definición de este método corresponden a los parámetros que se deben ingresar al instanciar un objeto.

```
def __init__(self, dni, nombre, apellido, sexo):
        """Constructor de clase Persona"""
```

```
self.dni = dni
self.nombre = nombre
self.apellido = apellido
self.sexo = sexo
```

Ahora para crear una nueva instancia de **Persona** cargando todos sus datos, simplemente tenemos que instanciar de la siguiente manera:

```
>>> pepe=Persona("25210458","Jose","Lopez","M")
```

Y habremos instanciado a **pepe** con todos sus datos. Seguramente, en este punto surge una duda: ¿qué pasa con el argumento **self**? No le enviamos nada al método y no se dispara ninguna excepción?

La respuesta es que **self** hace referencia al propio objeto que llama al método y que, aun sin ser especificado por nosotros, el primer parámetro que recibe el método es el propio objeto que lo llama.

VARIABLES DE CLASE Y DE INSTANCIA

En general, las variables de instancia son para datos únicos de cada instancia, y las variables de clase son para atributos y métodos compartidos por todas las instancias de la clase:

```
class Perro:

    tipo = 'canino'                 # variable de clase compar-
tida por todas las instancias

def __init__(self, nombre):
self.nombre = nombre        # variable de instancia única para
la instancia

>>> d = Perro('Fido')
>>> e = Perro('Buddy')
```

```
>>>d.tipo                    # compartido por todos los perros
  'canino'
>>>e.tipo                    # compartido por todos los perros
  'canino'
>>>d.nombre                  # único para d
  'Fido'
>>>e.nombre                  # único para e
  'Buddy'
```

Los datos compartidos pueden tener efectos inesperados que involucren objetos mutables, como listas y diccionarios. Por ejemplo, la lista **trucos** en el siguiente código no debería ser usada como variable de clase, porque una sola lista sería compartida por todas las instancias de **Perro**:

```
class Perro:

    trucos = []         # uso incorrecto de una variable de clase

def __init__(self, nombre):
self.nombre = nombre

defagregar_truco(self, truco):
self.trucos.append(truco)

>>> d = Perro('Fido')
>>> e = Perro('Buddy')
>>>d.agregar_truco('girar')
>>>e.agregar_truco('hacerse el muerto')
>>>d.trucos                # compartidos por todos los perros
inesperadamente
  ['girar', 'hacerse el muerto']
```

El diseño correcto de esta clase sería usando una variable de instancia:

```
class Perro:

def __init__(self, nombre):
```

```
self.nombre = nombre
self.trucos = []    # crea una nueva lista vacía para cada
perro

defagregar_truco(self, truco):
self.trucos.append(truco)

>>> d = Perro('Fido')
>>> e = Perro('Buddy')
>>>d.agregar_truco('girar')
>>>e.agregar_truco('hacerse el muerto')
>>>d.trucos
['girar']
>>>e.trucos
['hacerse el muerto']
```

DATOS PARA TENER EN CUENTA

Los atributos de datos tienen preferencia sobre los métodos con el mismo nombre; para evitar conflictos de nombre accidentales, que pueden causar errores difíciles de encontrar en programas grandes, es prudente usar algún tipo de convención que minimice las posibilidades de que se produzcan dichos conflictos. Algunas convenciones pueden ser: poner los nombres de métodos con mayúsculas, prefijar los nombres de atributos de datos con una pequeña cadena única (a lo mejor solo un guion bajo), o usar verbos para los métodos y sustantivos para los atributos.

Tanto los métodos como los usuarios (clientes) ordinarios de un objeto pueden hacer referencia a los atributos de datos. En otras palabras, las clases no se usan para implementar tipos de datos abstractos puros. De hecho, en Python no hay nada que exija ocultar datos; todo se basa en convención. La implementación de Python escrita en C puede ocultar por completo detalles de implementación y el control de acceso a un objeto si es necesario; esto se suele usar en extensiones a Python escritas en C.

Los clientes deben usar los atributos de datos con cuidado; estos pueden romper invariantes que mantienen los métodos si pisan los atributos de datos. Los clientes tienen la posibilidad de añadir sus propios atributos de datos a una instancia sin afectar la validez de sus métodos, siempre y cuando se eviten conflictos de nombres; una convención de nombres puede ahorrar muchos dolores de cabeza.

Herencia

¿QUÉ ES LA HERENCIA Y PARA QUÉ SIRVE?

Supongamos que tenemos la clase **Persona** con la que venimos trabajando para completar, por ejemplo, ciertos datos genéricos en una base de datos. Todas las personas tendrán las mismas características: nombre, DNI, edad, etcétera.

Ahora, imaginemos que debemos realizar un programa en el cual se incluyen diferentes tipos de personas, como por ejemplo: profesor, comerciante, etcétera.

Estos nuevos tipos de personas tendrán características compartidas, pero también poseerán características propias.

La herencia nos sirve para no repetir código que ya escribimos en una cierta clase y poder reutilizarlo en otra. Para ver cómo funcionaría esto, observemos el ejemplo práctico:

```
class Persona:
    """Clase que representa una Persona"""
    cedula = "V-13458796"
    nombre = "Leonardo"
    apellido = "Caballero"
    sexo = "M"
```

Si tuviéramos que modelar la clase **Profesor** sin tener previamente la clase **Persona**, nuestra definición sería así:

```
class Profesor:
    """Clase que representa un Profesor"""
    cedula = "V-13458796"
```

En el presente capítulo introduciremos un concepto fundamental, ya mencionado en el capítulo anterior, que nos facilitará considerablemente la vida en el momento de programar comportamientos similares.

03

```
    nombre = "Leonardo"
    apellido = "Caballero"
    sexo = "M"
cargosEnEscuela: 5
```

Si nos fijamos en el ejemplo presentado, el único atributo que hemos agregado es **cargosEnEscuela**, pero todo lo anterior es lo mismo a lo definido en la clase **Persona**.

Para evitar esta reescritura de código y facilitar el mantenimiento del código en el futuro, contamos con el concepto de **herencia**. De esta manera, podríamos decir que **Profesor** es una clase derivada de **Persona** y solo falta agregarle el comportamiento diferente:

```
class Profesor(Persona):
   """Clase que representa a un Profesor y deriva de Persona"
cargosEnEscuela: 5
```

Así, si queremos generalizar la declaración de clases mediante herencia, podríamos realizar el siguiente esquema:

```
class ClaseDerivada(ClaseBase):
<declaración-1>
    .
    .
    .
<declaración-N>
```

El nombre **ClaseBase** debe estar definido en un ámbito que contenga la definición de la clase derivada. En el lugar del nombre de la clase base se permiten otras expresiones arbitrarias. Esto puede ser útil, por ejemplo, cuando la clase base está definida en otro módulo:

```
class ClaseDerivada(modulo.ClaseBase):
```

La ejecución de una definición de clase derivada procede de la misma forma que una clase base. Cuando el objeto clase se construye, se tiene en cuenta la clase base. Esto se usa para resolver referencias a atributos: si un atributo solicitado no se encuentra en la clase, la búsqueda continúa por la clase base. Esta regla se aplica recursivamente si la clase base misma deriva de alguna otra clase.

No hay nada en especial en la instanciación de clases derivadas: **ClaseDerivada()** crea una nueva instancia de la clase. Las referencias a métodos se resuelven de la siguiente manera: se busca el atributo de clase correspondiente, descendiendo por la cadena de clases base si es necesario, y la referencia al método es válida si se entrega un objeto función.

Las clases derivadas pueden redefinir métodos de su clase base. Como los métodos no tienen privilegios especiales cuando llaman a otros métodos del mismo objeto, un método de la clase base que llame a otro método definido en la misma clase base puede terminar llamando a un método de la clase derivada que lo haya redefinido.

Python tiene dos funciones integradas que funcionan con herencia:

- **`sinstance()` se utiliza para verificar el tipo de una instancia.**

- **`isinstance(obj, int)` devuelve `True` solo si `obj.__class__` es `int` o alguna clase derivada de `int`.**

- **`issubclass()` se utiliza para comprobar herencia de clase: `issubclass(bool, int)` da `True`, ya que `bool` es una subclase de `int`. Sin embargo, `issubclass(float, int)` devuelve `False`, porque `float` no es una subclase de `int`.**

HERENCIA MÚLTIPLE

Una novedad que presenta Python frente a otros lenguajes de programación es que soporta una forma de **herencia múltiple**. Una definición de clase con múltiples clases base:

```
classClaseDerivada(Base1, Base2, Base3):
<declaración-1>
    .
    .
    .
<declaración-N>
```

En general, podemos pensar en la búsqueda de los atributos heredados de clases superiores de izquierda a derecha. Por ejemplo, en el caso anterior se buscaría primero en la `ClaseDerivada`; si no se encuentra, entonces se pasaría a la `Base1`, luego en forma recursiva se buscaría en las clases base de `Base1`, y solo si no se encuentra allí se pasaría a buscar en `Base2`, y se seguiría con ese esquema hasta completar la búsqueda.

Ahora bien, esta sería una explicación ordenada y sencilla, pero la realidad es un poco más compleja.

El orden de resolución de métodos va cambiando en forma dinámica para soportar las llamadas cooperativas a `super()`. En otros lenguajes de programación que también admiten herencia múltiple, este enfoque se conoce como "llámese al siguiente método" y tiene mayor peso que la llamada al superior, que se encuentra en lenguajes con solo herencia simple.

Ahora bien, ¿qué es esto del ordenamiento dinámico? ¿Por qué complicarnos la vida con esta metodología?

El ordenamiento dinámico es necesario porque todos los casos de herencia múltiple exhiben una o más **relaciones en diamante**.

Una relación en diamante se define cuando se puede llegar al menos a una de las clases base por distintos caminos desde la clase de que está más abajo. Por ejemplo, todas las clases heredan de `object`, lo cual significa que por medio de todas las clases base de una herencia múltiple podemos llegar a la clase `object`. Para evitar que las clases base sean accedidas más de una vez, el algoritmo dinámico hace que el orden de búsqueda sea lineal, de manera que se preserve el orden de izquierda a derecha especificado en cada clase, que se llame a cada clase

base una sola vez y que esta sea monótona, es decir que una clase pueda tener clases derivadas sin afectar el orden de precedencia de sus clases base.

VARIABLES PRIVADAS

Las variables privadas de instancia, que no pueden accederse excepto desde dentro de un objeto, no existen en Python. Sin embargo, hay una convención que se sigue en la mayoría del código Python: un nombre prefijado con un guion bajo (por ejemplo, **`_spam`**) debería tratarse como una parte no pública de la API –más allá de que sea una función, un método o un dato– y considerarse un detalle de implementación que está sujeto a cambios sin aviso.

Ya que existe un caso de uso válido para los identificadores privados de clase (a saber: colisión de nombres con nombres definidos en las subclases), hay un soporte limitado para este mecanismo. Cualquier identificador con la forma **`__spam`** (al menos dos guiones bajos al principio, como mucho un guion bajo al final) es textualmente reemplazado por **`_nombredeclase__spam`**, donde **`nombredeclase`** es el nombre de clase actual al que se le sacan guiones bajos del comienzo (si los tuviera). Se modifica el nombre del identificador sin importar su posición sintáctica, siempre y cuando ocurra dentro de la definición de una clase.

La modificación de nombres es útil para dejar que las subclases sobrescriban los métodos sin romper las llamadas a los métodos desde la misma clase. Por ejemplo:

```
class Mapeo:
def __init__(self, iterable):
self.lista_de_items = []
self.__actualizar(iterable)

def actualizar(self, iterable):
foritem in iterable:
self.lista_de_items.append(item)

    __actualizar = actualizar   # copia privada del actuali-
zar() original

classSubClaseMapeo(Mapeo):
```

```
def actualizar(self, keys, values):
        # provee una nueva signatura para actualizar()
        # pero no rompe __init__()
foritem in zip(keys, values):
self.lista_de_items.append(item)
```

Hay que aclarar que las reglas de modificación de nombres están diseñadas principalmente para evitar accidentes; es posible acceder o modificar una variable que es considerada como privada. Esto hasta puede resultar útil en circunstancias especiales, tales como en el depurador.

Notemos que el código pasado a **exec** o **eval()** no considera que el nombre de clase de la clase que invoca sea la clase actual; esto resulta parecido al efecto de la sentencia global, efecto que es de similar manera restringido a código que es compilado en conjunto. La misma restricción aplica a **getattr()**, **setattr()** y **delattr()**, así como cuando se referencia a **__dict__** directamente.

ITERADORES

Con seguridad, hemos notado que la mayoría de los objetos contenedores pueden ser recorridos usando una sentencia **for**:

```
for elemento in [1, 2, 3]:
print(elemento)
for elemento in (1, 2, 3):
print(elemento)
for clave in {‘uno’:1, ‘dos’:2}:
print(clave)
forcaracter in “123”:
print(caracter)
forlinea in open(“miarchivo.txt”):
print(linea, end=’’)
```

Este estilo de acceso es limpio, conciso y conveniente. El uso de **iteradores** está impregnado y unifica a Python. En bambalinas, la sentencia **for** llama

a **iter()** en el objeto contenedor. La función devuelve un objeto iterador que define el método **__next__()** que accede elementos en el contenedor de a uno por vez. Cuando no hay más elementos, **__next__()** levanta una excepción **StopIteration**, que le avisa al bucle del **for** que hay que terminar. Podemos llamar al método **__next__()** usando la función integrada **__next__()**. El siguiente ejemplo muestra cómo funciona todo esto:

```
>>> s = 'abc'
>>>it = iter(s)
>>>it
<iteratorobject at 0x00A1DB50>
>>>next(it)
'a'
>>>next(it)
'b'
>>>next(it)
'c'
>>>next(it)
Traceback (mostrecentcalllast):
  File "<stdin>", line 1, in <module>
next(it)
StopIteration
```

Habiendo visto la mecánica del protocolo de iteración, es fácil agregar comportamiento de iterador a nuestras clases. Definimos un método **__iter__()** que devuelva un objeto con un método **__next__()**. Si la clase define **__next__()**, entonces alcanza con que **__iter__()** devuelva **self:**

```
>>>class Reversa:
...     """Iterador para recorrer una secuencia marcha
atrás."""
...     def __init__(self, datos):
...         self.datos = datos
...         self.indice = len(datos)
...     def __iter__(self):
...         returnself
...     def __next__(self):
```

```
...         ifself.indice == 0:
...             raiseStopIteration
...         self.indice = self.indice - 1
...         returnself.datos[self.indice]
...
>>>rev = Reversa('spam')
>>>iter(rev)
<__main__.Reversa object at 0x00A1DB50>
>>>forchar in rev:
...     print(char)
...
m
a
p
s
```

GENERADORES

Los **generadores** son una simple y poderosa herramienta para crear iteradores. Se escriben como funciones regulares, pero usan la sentencia **`yield`** cuando quieren devolver datos. Cada vez que se llama **`next()`** sobre él, el generador continúa desde donde dejó, y recuerda todos los valores de datos y cuál fue la sentencia ejecutada última. Un ejemplo muestra que los generadores son muy fáciles de crear:

```
>>>def reversa(datos):
...     forindice in range(len(datos)-1, -1, -1):
...         yield datos[indice]
...
>>>for letra in reversa('golf'):
...     print(letra)
...
f
l
o
g
```

Todo lo que puede ser hecho con generadores también puede ser hecho con iteradores basados en clases, como se describe en la sección anterior. Lo que hace que los generadores sean tan compactos es que los métodos `__iter__()` y `__next__()` son creados automáticamente.

Otra característica clave es que las variables locales y el estado de la ejecución son guardados de manera automática entre llamadas. Esto hace que la función sea más fácil de escribir y quede mucho más claro que si lo hacemos usando variables de instancia, tales como `self.indice` y `self.datos`.

Además de la creación automática de métodos y de guardar el estado del programa, cuando los generadores terminan, automáticamente levantan `StopIteration`. Combinadas, estas características facilitan la creación de iteradores, y hacen que no se necesite más esfuerzo que escribir una función regular.

EXPRESIONES GENERADORAS

Algunos generadores simples pueden ser codificados concisamente como expresiones mediante el uso de una sintaxis similar a las listas por comprensión, pero con paréntesis en vez de corchetes. Estas expresiones se utilizan en situaciones en las que el generador es usado inmediatamente por una función que lo contiene. Las **expresiones generadoras** son más compactas pero menos versátiles que definiciones completas de generadores, y tienden a utilizar menos memoria que las listas por comprensión equivalentes.

Ejemplos:

```
>>> sum(i*i for i in range(10))                 # suma de
cuadrados
285

>>>xvec = [10, 20, 30]
>>>yvec = [7, 5, 3]
>>> sum(x*y forx,y in zip(xvec, yvec))         # producto
escalar
260
```

```
>>>frommathimport pi, sin
>>>tabla_de_senos = {x: sin(x*pi/180) for x in range(0, 91)}

>>>palabras_unicas = set(wordfor line in page  forword in
line.split())

>>>mejor_promedio = max((alumno.promedio, alumno.nombre) for
alumno in graduados)

>>> data = 'golf'
>>>list(data[i] for i in range(len(data) - 1, -1, -1))
['f', 'l', 'o', 'g']
```

REFERENCIAS DEL CAPÍTULO

Para este e-book se tomaron fragmentos de la documentación de Python Argentina como base: **http://docs.python.org.ar/tutorial/3/classes.html**.

Manejo de archivos

COMPRENDER ARCHIVOS

A pesar de que hoy en día es muy difícil encontrar a alguien que desconozca el concepto de **archivo**, no está de más decir que un archivo es una porción de información relacionada lógicamente que puede ser usada por programas de computadora y que se guarda en un medio de almacenamiento digital. Para acceder a él con fines de lectura o escritura, se utiliza una ruta única.

Tal como era de esperarse, Python tiene integrados varios módulos y funciones dedicados al manejo de archivos. A lo largo del capítulo, iremos recorriendo parte de estos con el fin de mostrar las operaciones fundamentales.

Para empezar con el manejo de archivos, presentaremos la función **open**, la cual devuelve un objeto **archivo**, del que a continuación mostraremos algunos de sus atributos y métodos. Esta función tiene varios parámetros –tres en Python 2 y ocho en Python 3, para ser más exactos–, pero solo nos vamos a enfocar en los primeros dos. El primer parámetro corresponde a la ruta (absoluta o relativa al directorio de trabajo actual) del archivo sobre el que se desea operar, y el segundo hace referencia al modo en el que este se abrirá, tal como se muestra en la siguiente tabla:

MODO	SIGNIFICADO
`r`	Abrir para lectura (por defecto).
`w`	Abrir para escritura. Si el archivo no existe, lo crea y, si no, lo trunca.
`x`	Abrir exclusivamente para creación. Falla si el archivo ya existe.
`a`	Abrir para escritura. Agrega contenido sobre el final si ya existía.
`b`	Modo binario.
`t`	Modo texto (por defecto).
`+`	Abrir para lectura y escritura.

En el presente capítulo introduciremos el concepto de archivo y aprenderemos a manejarlos con programación orientada a objetos.

04

Plasmemos lo aprendido en un ejemplo práctico. Para eso, lo primero que haremos es abrir un archivo, al que llamaremos **spoilerito.txt**, en modo escritura:

```
objetoArchivo = open("spoilerito.txt", "w")
```

A continuación, utilizando el método **write** escribiremos texto dentro del archivo de trabajo.

```
objetoArchivo.write("Pido perdón a los niños por haber dedi-
cado este libro a una persona mayor.\n" \
        + "...\n" \
        + "¡Sean amables con él! Y comuníquenme rápidamente
que ha regresado. ¡No me dejen tan triste!\n")
```

Es muy importante, sobre todo cuando se abre un archivo en modo escritura, cerrar el objeto **archivo** con el método **close** para evitar inconsistencias en los datos.

```
objetoArchivo.close()
```

Una buena práctica a la hora de leer archivos (y de programar en general) consiste en realizar un adecuado manejo de excepciones, de modo tal de no perder el control del flujo de nuestro programa. Esto es posible hacerlo insertando un código dentro de un bloque **try..finally** o bien podríamos operar el archivo utilizando la sentencia **with**, con lo que nuestro código pasaría a verse de la siguiente forma:

```
with open("spoilerito.txt", "w") as objetoArchivo:
    texto = "Pido perdón a los niños por haber dedicado este
libro a una persona mayor.\n" \
    + "...\n" \
    + "¡Sean amables con él! Y comuníquenme rápidamente que
ha regresado. ¡No me dejen tan triste!\n"
objetoArchivo.write(texto)
```

Lo notable de este método es que, por añadidura, nos desliga del método **close**, con todo lo que un posible olvido de dicho llamado implica.

Ahora veamos cómo abrir un archivo en modo lectura, en este caso el mismo que creamos antes. Esta vez a la llamada a la función **open** podemos asignarle o no el parámetro **mode** ya que por defecto se abre en modo lectura.

```
objetoArchivo = open("spoilerito.txt")
```

Para leer el archivo, vamos a utilizar el método **read** y almacenar su resultado dentro de la variable **texto**. Luego la imprimimos para ver su contenido.

```
with open("spoilerito.txt") as objetoArchivo:
     texto = objetoArchivo.read()
     print (texto)
```

Algo que aún no mencionamos sobre el método **read** es que tiene un parámetro opcional llamado **size**, que es un número entero que determina cuántos bytes deben ser leídos del archivo. En el caso de que no se le defina ningún valor, leerá el archivo hasta encontrar EOF (*end-of-file* o fin de fichero). Es decir que si quisiéramos, por ejemplo, leer únicamente los diez primeros bytes del archivo, tendríamos que hacerlo de la siguiente forma:

```
objetoArchivo.read(10)
```

Si al terminar esta operación volviéramos a utilizar el método **read** sobre el archivo, esta vez sin definir el parámetro **size**, obtendríamos la lectura de los bytes restantes, es decir, hasta que el cursor encuentre EOF.

Esta forma de leer un archivo nos lleva a pensar en la necesidad de saber en dónde se encuentra ubicado el cursor del objeto **archivo**, e incluso también, en poder asignarlo a cualquier posición de él. Tal como es de imaginarse, estos métodos existen y se llaman **tell** y **seek**, respectivamente. Veamos entonces con algún ejemplo práctico cómo se usan estos útiles y sencillos métodos:

```
with open("spoilerito.txt") as objetoArchivo:
     objetoArchivo.read(10)
     posCursor = objetoArchivo.tell()
     print (posCursor)
     objetoArchivo.seek(14)
     posCursor2 = objetoArchivo.tell()
     print (posCursor2)
```

Otra forma muy simple de leer un archivo consiste en realizar una iteración del tipo **foor..loop** para ir recorriéndolo línea por línea.

```
with open("spoilerito.txt") as objetoArchivo:
forlinea in objetoArchivo:
print(linea.rstrip())
```

Como podemos ver, en el ejemplo se utilizó el método **rstrip**, cuya función es la de eliminar los eventuales espacios al final de cada cadena línea.

A su vez, también podemos realizar una lectura por línea utilizando el método **readline**, el cual, tal como su nombre lo indica, lee una línea.

```
with open("spoilerito.txt") as objetoArchivo:
textoDeLinea = objetoArchivo.readline()
print(textoDeLinea)
```

Por último, existe también el método **readlines**, que devuelve una lista compuesta por una cadena por cada línea contenida en el archivo.

```
with open("spoilerito.txt") as objetoArchivo:
listaDeTextosDeLineas = objetoArchivo.readlines()
```

Módulos y biblioteca estándar

MÓDULOS

Como ya hemos visto, si quisiéramos escribir un programa más o menos largo, lo ideal sería hacerlo en un archivo y, luego, correr dicho archivo en el intérprete de Python. Esto se conoce como **ejecutar un script**.

Ahora bien, ¿qué pasa si el programa es muy extenso o si parte de ese programa nos sirve para otros proyectos?

Para soportar esto, Python tiene una manera de poner definiciones en un archivo y usarlas en un script o en una instancia interactiva del intérprete. Tal archivo es llamado **módulo**, y las definiciones de un módulo pueden ser importadas a otros módulos o al módulo principal (la colección de variables a las que se tiene acceso en un script ejecutado en el nivel superior y en el modo calculadora).

Un módulo es un archivo que contiene definiciones y declaraciones de Python. El nombre del archivo es el nombre del módulo con el sufijo `.py` agregado. Dentro de un módulo, su nombre (como una cadena) está disponible en el valor de la variable global `__name__`. Por ejemplo, podríamos crear un archivo llamado `fibo.py` en el directorio actual con el siguiente contenido:

```
# módulo de números Fibonacci

deffib(n):    # escribe la serie Fibonacci hasta n
    a, b = 0, 1
while b < n:
print(b, end=’ ‘)
        a, b = b, a+b
print()

def fib2(n):   # devuelve la serie Fibonacci hasta n
    resultado = []
```

Habiendo visto las bases de la programación orientada a objetos, resulta pertinente comprender qué es un módulo y realizar un recorrido por la biblioteca estándar de Python, para conocer a fondo las herramientas con las que contamos en el momento de comenzar nuestros propios proyectos.

05

```
    a, b = 0, 1
while b < n:
resultado.append(b)
        a, b = b, a+b
return resultado
```

Ahora ingresamos al intérprete de Python e importamos este módulo con la siguiente orden:

```
>>>importfibo
```

Entonces, contaremos con los nombres de las funciones definidas en **fibo** en el espacio de nombres actuales, pero solo llamándolas con el prefijo **fibo**:

```
>>>fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>>fibo.__name__
'fibo'
```

No obstante, es posible asignar un nombre local a una función. Esto se justifica en el caso de que la función se utilice con frecuencia:

```
>>>fib = fibo.fib
>>>fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

MÁS SOBRE LOS MÓDULOS

Un módulo puede contener tanto declaraciones ejecutables como definiciones de funciones. Estas declaraciones están pensadas para inicializar el módulo. Se ejecutan solo la primera vez que el módulo se encuentra en una sentencia **import**.

Cada módulo tiene su propio espacio de nombres, que es usado como espacio de nombres global por todas las funciones definidas en el módulo. Por lo tanto, el autor de un módulo puede usar variables globales en el módulo sin preocuparse acerca de conflictos con una variable global del usuario.

Los módulos pueden importar otros módulos. Se acostumbra ubicar todas las declaraciones **import** al principio del módulo, aunque no es obligatorio que se ubiquen en ese lugar. Los nombres de los módulos importados se colocan en el espacio de nombres global del módulo que hace la importación.

Hay una variante de la declaración **import** que importa los nombres de un módulo directamente al espacio de nombres del módulo que hace la importación. Por ejemplo:

```
>>>fromfiboimportfib, fib2
>>>fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

En este caso, no se está importando el módulo completo, por lo que no sería válida la siguiente expresión:

```
>>>fibo.fib(1000)
```

y se produciría una excepción.

En general, se considera una mala práctica realizar un **import** * de un módulo o paquete, ya que el código en sí no queda bien definido o legible y podríamos estar importando funciones que ni siquiera utilizaremos.

Por razones de eficiencia, cada módulo se importa una vez por sesión del intérprete. Por lo tanto, si se modifican los módulos, sería necesario reiniciar el intérprete.

EJECUTAR MÓDULOS COMO SCRIPTS

Cuando se ejecuta un módulo de Python con

```
python fibo.py <argumentos>
```

se ejecutará el código en el módulo tal como si lo hubiésemos importado, pero con **__name__** con el valor de **"__main__"**. Eso significa que si agregamos este código al final de dicho módulo:

```
if __name__ == "__main__":
importsys
fib(int(sys.argv[1]))
```

es posible que el archivo sea utilizable tanto como script, como módulo importable, porque el código que analiza la línea de órdenes solo se ejecuta si el módulo es ejecutado como archivo principal:

```
$ python fibo.py 50
1 1 2 3 5 8 13 21 34
```

En cambio, si el módulo se importa, ese código no se ejecuta:

```
>>>importfibo
>>>
```

Esto suele usarse para proveer al módulo de una interfaz de usuario conveniente o para propósitos de prueba, ya que ejecutar el módulo como un script ejecuta el juego de pruebas.

EL CAMINO DE BÚSQUEDA DE LOS MÓDULOS

Cuando se importa un módulo llamado **spam**, el intérprete busca primero un módulo con ese nombre, que esté integrado en el intérprete. Si no lo encuentra, entonces busca un archivo llamado `spam.py` en una lista de directorios especificada por la variable `sys.path`.

En sistemas de archivos que soportan enlaces simbólicos, el directorio que contiene el script de entrada es calculado después de que el enlace simbólico sea seguido. En otras palabras, el directorio que incluye el enlace simbólico no es agregado al módulo de busca del path.

Luego de la inicialización, los programas Python pueden modificar `sys.path`. El directorio que contiene el script que se está ejecutando se ubica al principio de la búsqueda, adelante de la biblioteca estándar. Esto significa que se cargarán scripts en ese directorio en lugar de módulos de la biblioteca estándar con el mismo nombre. Esto es un error a menos que se esté reemplazando intencionalmente.

MÓDULOS ESTÁNDARES

Python viene con una biblioteca de módulos estándares. Algunos módulos se integran en el intérprete; estos proveen acceso a operaciones que no son parte del núcleo del lenguaje, tanto por eficiencia como para proveer acceso a primitivas del sistema operativo y a llamadas al sistema. El conjunto de tales módulos es

Archivos compilados de Python

Para acelerar la carga de módulos, Python cachea las versiones compiladas de cada módulo en el directorio `__pycache__` bajo el nombre `module.version.pyc`; en él, la versión codifica el formato del archivo compilado y generalmente contiene el número de versión de Python. Por ejemplo, en *CPythonrelease 3.3* la versión compilada de spam.py sería cacheada como `__pycache__/spam.cpython-33.pyc`. Esta convención de nombre permite compilar módulos desde diferentes releases y versiones de Python para coexistir.

una opción de configuración que también depende de la plataforma subyacente. Por ejemplo, el módulo **winreg** solo se provee en sistemas Windows. Un módulo en particular merece algo de atención: **sys**, que se encuentra integrado en todos los intérpretes de Python. Las variables **sys.ps1** y **sys.ps2** definen las cadenas usadas como cursores primarios y secundarios:

```
>>>importsys
>>> sys.ps1
‘>>> ‘
>>> sys.ps2
‘... ‘
>>> sys.ps1 = ‘C> ‘
C>print(‘Yuck!’)
Yuck!
C>
```

Estas dos variables solo están definidas si el intérprete está en modo interactivo.

La variable **sys.path** es una lista de cadenas que determinan el camino de búsqueda del intérprete para los módulos. Se inicializa por omisión a un camino tomado de la variable de entorno **PYTHONPATH**, o a un valor predefinido en el intérprete si **PYTHONPATH** no está configurada. Es posible modificarlo usando las operaciones estándares de listas:

```
>>>importsys
>>>sys.path.append(‘/ufs/guido/lib/python’)
```

INTERFAZ AL SISTEMA OPERATIVO

El módulo **os** provee de un gran número de funciones para interactuar con el sistema operativo:

```
>>>import os
>>>os.getcwd()      # devuelve el directorio de trabajo actual
‘C:\\Python36’
```

```
>>>os.chdir('/server/accesslogs')   # cambia el directorio de
trabajo
>>>os.system('mkdirtoday')   # ejecuta el comando 'mkdir' en
el sistema
0
```

Las funciones integradas **dir()** y **help()** son útiles como ayudas interactivas para trabajar con módulos grandes como **os**:

```
>>>import os
>>>dir(os)
<devuelve una lista de todas las funciones del módulo>
>>>help(os)
<devuelve un manual creado a partir de las documentaciones
del módulo>
```

Para tareas diarias de administración de archivos y directorios, el módulo **shutil** provee una interfaz de más alto nivel que es más fácil de usar:

```
>>>importshutil
>>>shutil.copyfile('datos.db', 'archivo.db')
'archivo.db'
>>>shutil.move('/build/executables', 'dir_instalac')
'dir_instalac'
```

COMODINES DE ARCHIVOS

El módulo **glob** provee una función para hacer listas de archivos a partir de búsquedas con comodines en directorios:

```
>>>importglob
>>>glob.glob('*.py')
['primes.py', 'random.py', 'quote.py']
```

ARGUMENTOS DE LÍNEA DE ÓRDENES

Los programas frecuentemente necesitan procesar argumentos de línea de órdenes. Estos argumentos se almacenan en el atributo **argv** del módulo **sys** como una lista. Por ejemplo, la siguiente salida resulta de ejecutar **python demo.py uno dos tres** en la línea de órdenes:

```
>>>importsys
>>>print(sys.argv)
['demo.py', 'uno', 'dos', 'tres']
```

El módulo **getopt** procesa **sys.argv** usando las convenciones de la función de Unix **getopt()**. El módulo **argparse** provee un procesamiento más flexible de la linea de órdenes.

REDIRECCIÓN DE LA SALIDA DE ERROR Y FINALIZACIÓN DEL PROGRAMA

El módulo **sys** también tiene atributos para **stdin**, **stdout** y **stderr**. Este último es útil para emitir mensajes de alerta y error para que se vean, incluso, cuando se haya redireccionado **stdout**:

```
>>>sys.stderr.write('Alerta, archivo de log no
encontrado\n')
Alerta, archivo de log no encontrado
```

La forma más directa de terminar un programa es usar **sys.exit()**.

COINCIDENCIA EN PATRONES DE CADENAS

El módulo **re** provee herramientas de expresiones regulares para un procesamiento avanzado de cadenas. Para manipulación y coincidencias complejas, las expresiones regulares ofrecen soluciones concisas y optimizadas:

```
>>>import re
>>>re.findall(r'\bt[a-z]*', 'tres felices tigres comen trigo')
['tres', 'tigres', 'trigo']
>>>re.sub(r'(\b[a-z]+) \1', r'\1', 'gato en el el sombrero')
'gato en el sombrero'
```

Cuando se necesita algo más sencillo, se prefieren los métodos de las cadenas porque son más fáciles de leer y depurar.

```
>>> 'te para tos'.replace('tos', 'dos')
'te para dos'
```

MATEMÁTICA

El módulo **math** permite el acceso a las funciones de la biblioteca C subyacente para la matemática de punto flotante:

```
>>>importmath
>>>math.cos(math.pi / 4)
0.70710678118654757
>>> math.log(1024, 2)
10.0
```

El módulo **random** provee herramientas para realizar selecciones al azar:

```
>>>importrandom
>>>random.choice(['manzana', 'pera', 'banana'])
'manzana'
>>>random.sample(range(100), 10)   # elección sin reemplazo
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>>random.random()    # un float al azar
0.17970987693706186
>>>random.randrange(6)    # un entero al azar tomado de
range(6)
4
```

El módulo **statistics** calcula propiedades de estadística básica (la media, mediana, varianza, etcétera) de datos numéricos:

```
>>>importstatistics
>>> datos = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5]
>>>statistics.mean(datos)
1.6071428571428572
>>>statistics.median(datos)
1.25
>>>statistics.variance(datos)
1.3720238095238095
```

El proyecto **SciPy** (https://scipy.org) tiene muchos otros módulos para cálculos numéricos.

ACCESO A INTERNET

Hay varios módulos para acceder a internet y procesar sus protocolos. Dos de los más simples son **urllib.request** para traer data de URLs y **smtplib** para mandar correos:

```
>>>withurlopen('http://tycho.usno.navy.mil/cgi-bin/timer.pl')
as response:
...     for line in response:
...         line = line.decode('utf-8')  # Decodingthebinary
data totext.
...         if 'EST' in line or 'EDT' in line:  # look for
Eastern Time
...             print(line)

<BR>Nov. 25, 09:43:32 PM EST

>>>importsmtplib
>>> server = smtplib.SMTP('localhost')
>>>server.sendmail('soothsayer@ejemplo.org', 'jcaesar@ejemplo.
```

```
org',
... """To: jcaesar@ejemplo.org
... From: soothsayer@ejemplo.org
...
... Ojo al piojo.
... """)
>>>server.quit()
```

FECHAS Y TIEMPOS

El módulo **datetime** ofrece clases para manejar fechas y tiempos tanto de manera simple como compleja. Aunque soporta aritmética sobre fechas y tiempos, el foco de la implementación se ubica en la extracción eficiente de partes para manejarlas o formatear la salida. El módulo también soporta objetos que son conscientes de la zona horaria.

```
>>> # las fechas son fácilmente construidas y formateadas
>>>fromdatetimeimport date
>>> hoy = date.today()
>>> hoy
datetime.date(2009, 7, 19)

>>> # nos aseguramos de tener la info de localización cor-
recta
>>>importlocale
>>>locale.setlocale(locale.LC_ALL, locale.getdefaultlocale())
'es_ES.UTF8'
>>>hoy.strftime("%m-%d-%y. %d %b %Y es %A. hoy es %d de %B.")
'07-19-09. 19 jul 2009 es domingo. hoy es 19 de julio.'

>>> # las fechas soportan aritmética de calendario
>>> nacimiento = date(1964, 7, 31)
>>> edad = hoy - nacimiento
>>>edad.days
14368
```

COMPRESIÓN DE DATOS

Los formatos para archivar y comprimir datos se soportan directamente con los módulos: **zlib**, **gzip**, **bz2**, **lzma**, **zipfile** y **tarfile**.

```
>>>importzlib
>>> s = b'witchwhich has whichwitcheswristwatch'
>>>len(s)
41
>>> t = zlib.compress(s)
>>>len(t)
37
>>>zlib.decompress(t)
b'witchwhich has whichwitcheswristwatch'
>>> zlib.crc32(s)
226805979
```

MEDICIÓN DE RENDIMIENTO

Algunos usuarios de Python desarrollan un profundo interés en saber el rendimiento relativo de las diferentes soluciones al mismo problema. Python provee una herramienta de medición que responde esas preguntas inmediatamente.

Por ejemplo, puede ser tentador usar la característica de empaquetamiento y desempaquetamiento de las tuplas en lugar de la solución tradicional para intercambiar argumentos. El módulo **timeit** muestra rápidamente una modesta ventaja de rendimiento:

```
>>>fromtimeitimportTimer
>>>Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()
0.57535828626024577
>>>Timer('a,b = b,a', 'a=1; b=2').timeit()
0.54962537085770791
```

En contraste con el fino nivel de granularidad de este módulo, los módulos **profile** y **pstats** proveen herramientas para identificar secciones críticas de tiempo en bloques de código más grandes.

CONTROL DE CALIDAD

Una forma para desarrollar software de alta calidad es escribir pruebas para cada función mientras se la desarrolla, y correr esas pruebas frecuentemente durante el proceso de desarrollo. El módulo **doctest** provee una herramienta para revisar un módulo y validar las pruebas integradas en las cadenas de documentación (o docstring) del programa. La construcción de las pruebas es tan sencilla como cortar y pegar una ejecución típica junto con sus resultados en los docstrings. Esto mejora la documentación al proveer al usuario de un ejemplo y permite que el módulo **doctest** se asegure de que el código permanezca fiel a la documentación:

```
def promedio(valores):
    """Calcula la media aritmética de una lista de números.

>>>print(promedio([20, 30, 70]))
    40.0
    """
return sum(valores) / len(valores)
```

El módulo **unittest** necesita más esfuerzo que el módulo **doctest**, pero permite que se mantenga, en un archivo separado, un conjunto más comprensivo de pruebas:

```
importunittest

classTestFuncionesEstadisticas(unittest.TestCase):

deftest_promedio(self):
self.assertEqual(promedio([20, 30, 70]), 40.0)
self.assertEqual(round(promedio([1, 5, 7]), 1), 4.3)
withself.assertRaises(ZeroDivisionError):
          promedio([])
withself.assertRaises(TypeError):
          promedio(20, 30, 70)
```

unittest.main() **#**, llamado de la línea de comandos, ejecuta todas las pruebas.

LAS PILAS INCLUIDAS

Python tiene una filosofía de "pilas incluidas". Esto se ve mejor en las capacidades robustas y sofisticadas de sus paquetes más grandes.

Por ejemplo: los módulos **xmlrpc.client** y **xmlrpc.server** hacen que implementar llamadas a procedimientos remotos sea una tarea trivial. A pesar de los nombres de los módulos, no se necesita conocimiento directo o manejo de XML.

El paquete **email** es una biblioteca para manejar mensajes de e-mail, incluyendo MIME y otros mensajes basados en RFC 2822. Al contrario de **smtplib** y **poplib**, que en realidad envían y reciben mensajes, el paquete **email** tiene un conjunto de herramientas completo para construir y decodificar estructuras complejas de mensajes (incluyendo adjuntos) y para implementar protocolos de cabecera y codificación de internet.

El paquete **json** provee soporte robusto para parsear este popular formato de intercambio de datos. El módulo **csv** soporta una lectura y escritura directa de archivos de valores separados por coma, comúnmente soportados por bases de datos y hojas de cálculos. Procesar XML es soportado por los paquetes **xml.etree.ElementTree**, **xml.dom** y **xml.sax**. Juntos, estos tres módulos y paquetes simplifican ampliamente el intercambio de datos entre aplicaciones Python y otras herramientas.

El módulo **sqlite3** es un wrapper para la librería de base de datos SQLite; provee una base de datos persistente que puede ser actualizada y accedida utilizando sintaxis SQL ligeramente no estándar.

Se soporta la internacionalización a través de varios módulos, incluyendo **gettext**, **locale**, y el paquete **codecs**.

Errores comunes y buenas prácticas

ROMPER CÓDIGOS

Luego de haber pasado por un capítulo en el que desarrollamos y testeamos varios códigos diferentes, con seguridad nos toparemos más de una vez con algún error que nos impida ejecutar correctamente el código en el primer intento. De hecho, en el mundo de la programación, es una utopía pensar que el código que escribimos puede funcionar de entrada sin ningún error.

Sobre esta base, retomaremos los códigos que hemos desarrollado antes e iremos recorriendo todos los posibles errores que nos pueden surgir a la hora de querer correrlos, intérprete de por medio.

ERRORES AL INTENTAR EJECUTAR UN SCRIPT DE PYTHON

Uno de los principales errores que es posible cometer si queremos utilizar archivos para realizar nuestros códigos es intentar abrir estos archivos desde el intérprete interactivo.

```
>>> python3 holaMundo.py
```

Esto resultará en un bonito código de error que podemos ir sumando a la lista de errores que veremos en el presente capitulo.

Siempre que queramos ejecutar un archivo (de ahora en adelante script) de Python, tenemos que hacerlo desde la consola y no desde el intérprete interactivo.

En este capítulo romperemos todos los códigos de ejemplo del capítulo anterior, para analizar uno por uno los posibles errores que podemos cometer al escribir nuestros códigos. Además, ampliaremos sobre las buenas prácticas en programación y analizaremos diferentes formas de aplicarlas.

06

```
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
celeste@cgging:~$ python3
Python 3.6.7 (default, Oct 22 2018, 11:32:17)
[GCC 8.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> python3 holaMundo.py
  File "<stdin>", line 1
    python3 holaMundo.py
                   ^
SyntaxError: invalid syntax
>>>
```

HOLA MUNDO "CRUEL"

El primer ejemplo que vimos fue el mundialmente famoso y nunca bien ponderado "Hola Mundo". En este apartado nos referiremos más bien al "Mundo cruel" ya que, si bien este ejemplo es muy sencillo por tratarse de una sola línea de código, es probable que no hayamos podido correrlo con éxito en el primer intento.

¿Con que nos podemos encontrar en este caso? Para comenzar, en este ejemplo habíamos introducido el uso de archivos para almacenar el código en lugar de escribirlo directamente en el intérprete interactivo, y aquí un error común es omitir la extensión del archivo al ejecutar el código:

```
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
celeste@cgging:~$ python3 holaMundo.py
Hola Mundo
celeste@cgging:~$ clear
celeste@cgging:~$ python3 holaMundo
python3: can't open file 'holaMundo': [Errno 2] No such file or directory
celeste@cgging:~$
```

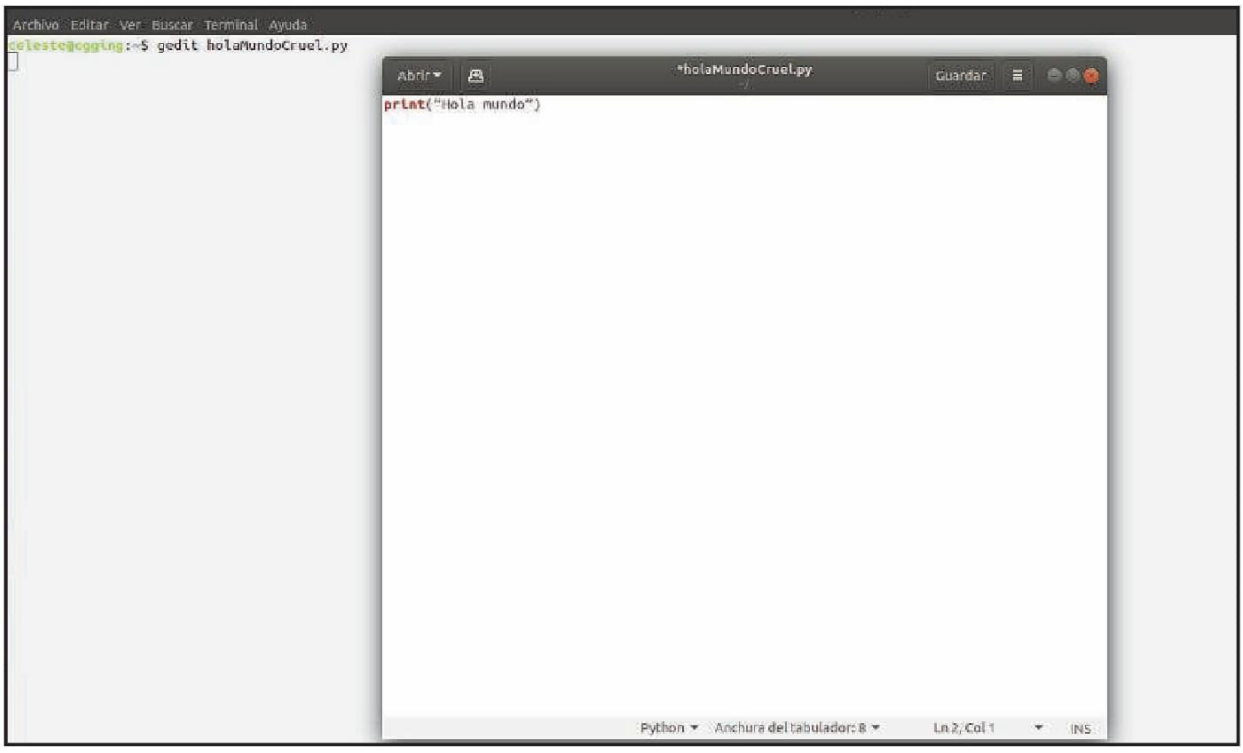
Otro error puede ser omitir mayúsculas en el nombre, recordemos que nuestro archivo se llamaba **holaMundo.py**:

```
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
celeste@cgging:~$ python3 holamundo.py
python3: can't open file 'holamundo.py': [Errno 2] No such file or directory
celeste@cgging:~$
```

Por último, un error no tan trivial y que puede llegar a complicarnos bastante si no sabemos de qué se trata tiene que ver con el copy/paste del código que queremos ejecutar desde algún documento, tal como un archivo pdf hacia el archivo **.py** que estamos escribiendo.

```
print(“Hola mundo”)
```

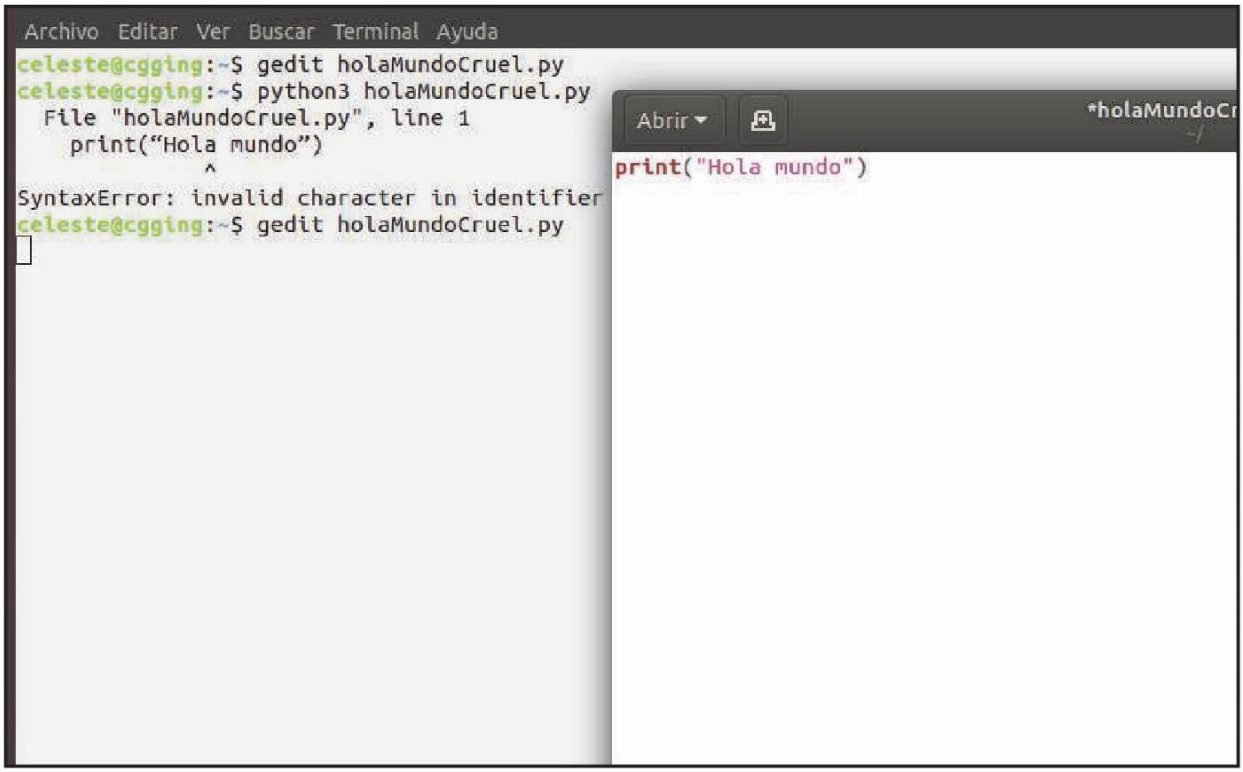
Si copiamos desde un archivo de texto el código que acabamos de escribir en este documento:



A simple vista –si es uno de los primeros códigos que realizamos o si utilizamos un editor de texto plano–, no seremos capaces de notar que este código no funcionará, pero al correrlo desde la consola:

```
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
celeste@cgging:~$ gedit holaMundoCruel.py
celeste@cgging:~$ python3 holaMundoCruel.py
  File "holaMundoCruel.py", line 1
    print(“Hola mundo”)
                ^
SyntaxError: invalid character in identifier
celeste@cgging:~$
```

El código no se ejecuta y nos aparece un error de sintaxis. ¿Qué ocurrió? El código de las comillas del texto que copiamos es diferente a las comillas que entiende Python, y si utilizamos Linux o algún editor de código que nos identifique las diferentes partes del código mediante colores, entonces podremos evidenciar que, al reemplazar las comillas en el documento, el texto del **`print`** cambiará de color.



Si ahora volvemos a correr el código, el error se habrá solucionado.

SUMA

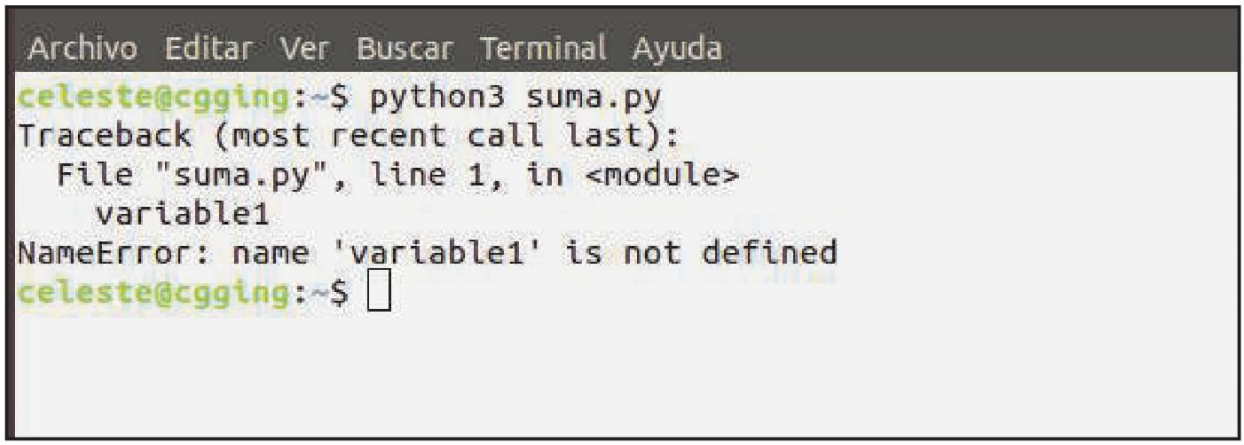
Analicemos el código que realizamos y comencemos a ver los posibles puntos de falla:

```
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)
```

El primer error que podríamos cometer al pensar este código, sobre todo si venimos de programar en algún otro lenguaje, es el intento de declarar las variables sin inicializarlas:

```
variable1
variable2
```

Esto producirá un error tal como habíamos mencionado en capítulos anteriores. Recordemos que las variables en Python se definen al asignarles un valor, ya que el tipado es dinámico.



Otro lugar en donde podemos cometer errores es en el **print**. En particular en este caso, queremos mostrar mucha información combinada, tanto de variables como de texto, y sería algo normal que alguna comilla se nos pasara de largo:

```
print(“la suma de la variable 1= , variable1, “ y la vari-
able 2=”, variable2, “es igual a “, suma)
```

En este caso, faltaría cerrar la comilla luego del =, omisión que producirá un error.

```
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
celeste@cgging:~$ python3 suma.py
  File "suma.py", line 4
    print("la suma de la variable1=,variable1," y la variable2=", variable2," es igual a ",suma)
                                                ^
SyntaxError: invalid syntax
celeste@cgging:~$
```

EXCEPCIONES

Como hemos mencionado brevemente en capítulos anteriores y como hemos visto en particular en el ejemplo “Suma con vitaminas”, es posible que nuestro código falle no por errores de sintaxis, sino por errores en tiempo de ejecución.

En el ejemplo que mencionamos, el error que se producía tenía que ver con el tipo de datos que ingresara el usuario. Estos errores se conocen como **excepciones** y es posible tomar recaudos para manejarlos.

Veamos algunos ejemplos de errores en tiempo de ejecución:

```
>>> 10 * (1/0)
Traceback (mostrecentcalllast):
  File “<stdin>”, line 1, in <module>
ZeroDivisionError: divisionbyzero
>>> 4 + spam*3
Traceback (mostrecentcalllast):
  File “<stdin>”, line 1, in <module>
NameError: name ‘spam’ isnotdefined
>>> ‘2’ + 2
Traceback (mostrecentcalllast):
  File “<stdin>”, line 1, in <module>
TypeError: Can’tconvert ‘int’ objecttostrimplicitly
```

La información que nos provee el intérprete en la última línea de los mensajes de error se corresponde con lo que ha ocurrido. Como podemos suponer, las excepciones tienen diferente tipo, y este se muestra como parte del mensaje de error (`ZeroDivisionError`, `NameError`, `TypeError`, etcétera).
El nombre del tipo de excepción mostrada coincide con el nombre de la excepción predefinida que ocurrió. La coincidencia en la descripción y el nombre se utiliza por un tema de convención, no obstante, nosotros podemos definir nuestras propias excepciones y en ese caso mostraríamos un texto diferente del nombre de la excepción por pantalla, aunque una buena práctica es mantener iguales el texto mostrado y el nombre, para facilitar la tarea de depuración. Los nombres de las excepciones estándares son identificadores incorporados al intérprete (no son palabras clave reservadas).

El resto de la línea provee un detalle basado en el tipo de la excepción y qué la causó.

La parte anterior del mensaje de error muestra el contexto donde la excepción sucedió, en la forma de un trazado del error en forma de listas de las líneas fuente; sin embargo, no mostrará líneas leídas desde la entrada estándar.

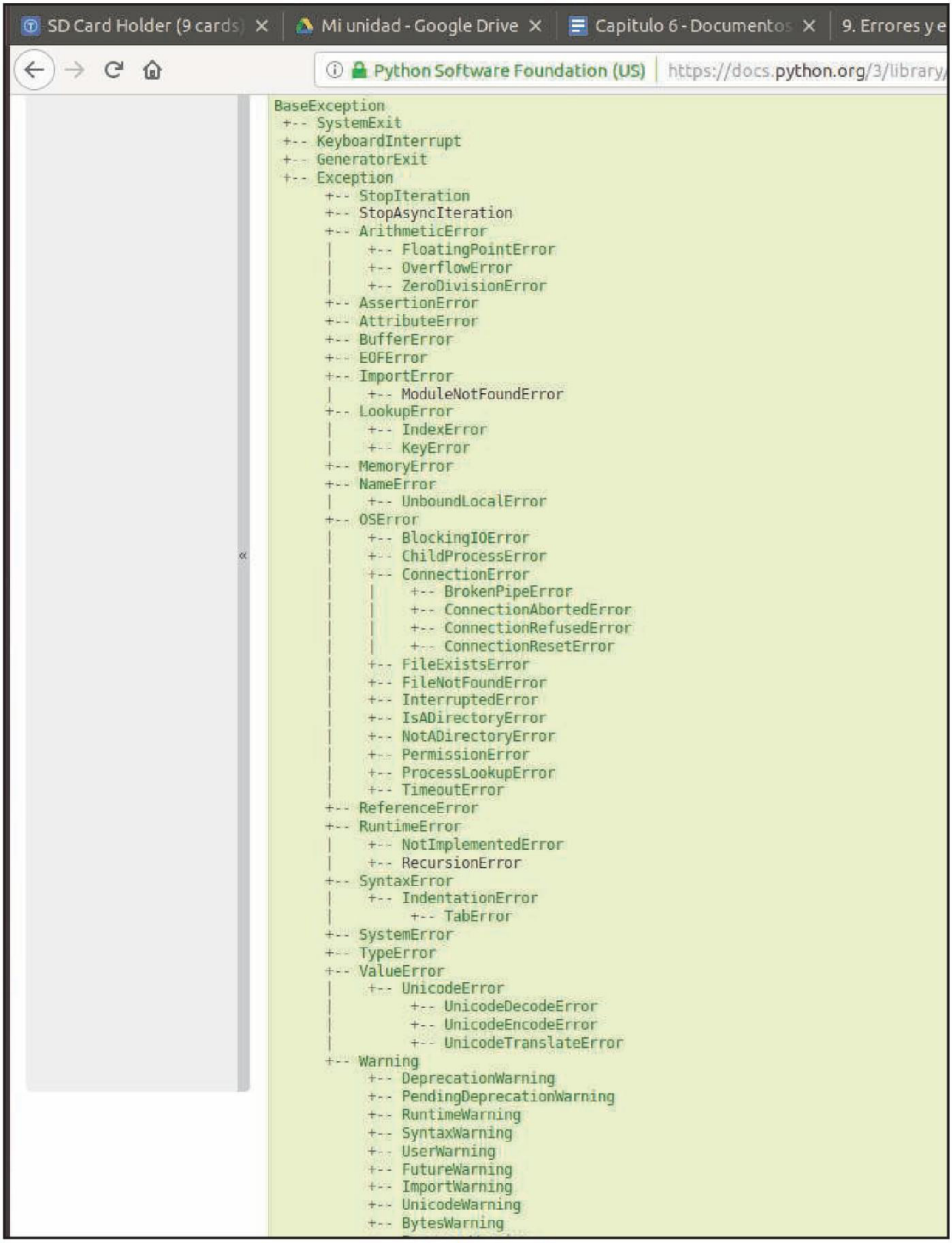
MANEJAR EXCEPCIONES

Si las excepciones son errores en tiempo de ejecución, pero tienen un nombre definido, es natural pensar que conociendo el posible error que ocurrirá podremos manejar esa situación.

Veamos un ejemplo:

```
while True:
    try:
     valor1 = int(input("Por favor ingrese el primer nu-
mero: "))
     break
     exceptValueError:
     print("Oops! No era válido. Intente nuevamente...")
```

En el ejemplo anterior, notamos que se introdujo una nueva declaración que encierra el bloque que podría tener el error en tiempo de ejecución, como habíamos visto en el caso de "Suma con Vitaminas", donde el error se producía



Listado de excepciones integradas en el intérprete de Python.

al querer ingresar un dato no numérico por teclado. Por lo tanto, deberemos encerrar con **try** esa porción de código.

La función del bloque **try** intenta la ejecución del código que encierra. Si no se produce ninguna excepción durante la ejecución, entonces el bloque **except** se saltea y se continua con el flujo normal del programa.

Si en cambio se produce una excepción, se interrumpirá el flujo del programa y se salteará el resto del código hasta pasar al bloque except, que se ejecutará en el caso de que la excepción ocurrida coincida con la manejada por este bloque, y luego es retornada nuevamente al comienzo del bloque **try**.

En el caso de que ocurriera una excepción diferente de la manejada por el bloque **except**, se busca si hay alguna declaración **try** de orden superior (es decir, algún **try** anidado por encima del bloque que produjo la excepción); si no se encuentra ningún bloque que la maneje, estamos frente a una excepción no manejada, y la ejecución se frena con el mensaje de error.

```
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
celeste@cgging:~$ gedit excepcion
celeste@cgging:~$ gedit excepcion.py
celeste@cgging:~$ python3 excepcion.py
Por favor ingrese el primer numero: r
Oops! No era válido. Intente nuevamente...
Por favor ingrese el primer numero: yu
Oops! No era válido. Intente nuevamente...
Por favor ingrese el primer numero: 4
celeste@cgging:~$
```

Una declaración try puede tener más de un **except**, para especificar manejadores para distintas excepciones. A lo sumo, un manejador será ejecutado. Solo se manejan excepciones que ocurren en el correspondiente **try**, no en otros manejadores del mismo try. Un **except** puede nombrar múltiples excepciones usando paréntesis, por ejemplo:

```
... except (RuntimeError, TypeError, NameError):
...  pass
```

Esto lo veremos con mayor detalle en el próximo e-book.

Por lo pronto, habiendo comprendido el uso de excepciones, las manejaremos en el ejemplo de “Suma con vitaminas”:

```
while True:
     try:
         variable1=int(input("Ingrese el primer valor:"))
         break
     exceptValueError:
     print("El valor debe ser numerico, por favor intente
nuevamente")
while True:
     try:
         variable2=int(input("Ingrese el segundo valor:"))
         break
     exceptValueError:
     print("El valor debe ser numerico, por favor intente
nuevamente")
suma= variable1 + variable2
print("la suma de la variable 1=" , variable1, " y la vari-
able 2=", variable2, "es igual a ", suma)
```

Ahora que hemos manejado la excepción que se produce cuando intentamos ingresar un valor no numérico, hemos limitado de manera considerable las fallas que se podrían producir en tiempo de ejecución con nuestro código.

```
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
celeste@cgging:~$ python3 sumaConExcep.py
Ingrese el primer valor:r
El valor debe ser numerico, por favor intente nuevamente
Ingrese el primer valor:[
El valor debe ser numerico, por favor intente nuevamente
Ingrese el primer valor:=
El valor debe ser numerico, por favor intente nuevamente
Ingrese el primer valor:
El valor debe ser numerico, por favor intente nuevamente
Ingrese el primer valor:45
Ingrese el segundo valor:d
El valor debe ser numerico, por favor intente nuevamente
Ingrese el segundo valor:13
la suma de la variable 1= 45  y la variable 2= 13 es igual a  58
celeste@cgging:~$
```

Analicemos un poco lo que acabamos de implementar. Realizamos un **while True** y un **try** por cada variable. Seguramente se nos ocurriría pensar que podríamos simplificarlo utilizando un solo bucle, ya que al tratarse de la misma posible excepción para las dos variables, si el usuario se equivocase al ingreso de cualquiera de las dos, se manejaría la excepción de igual manera. Veamos qué sucede si implementamos nuestro código con un solo **while**:

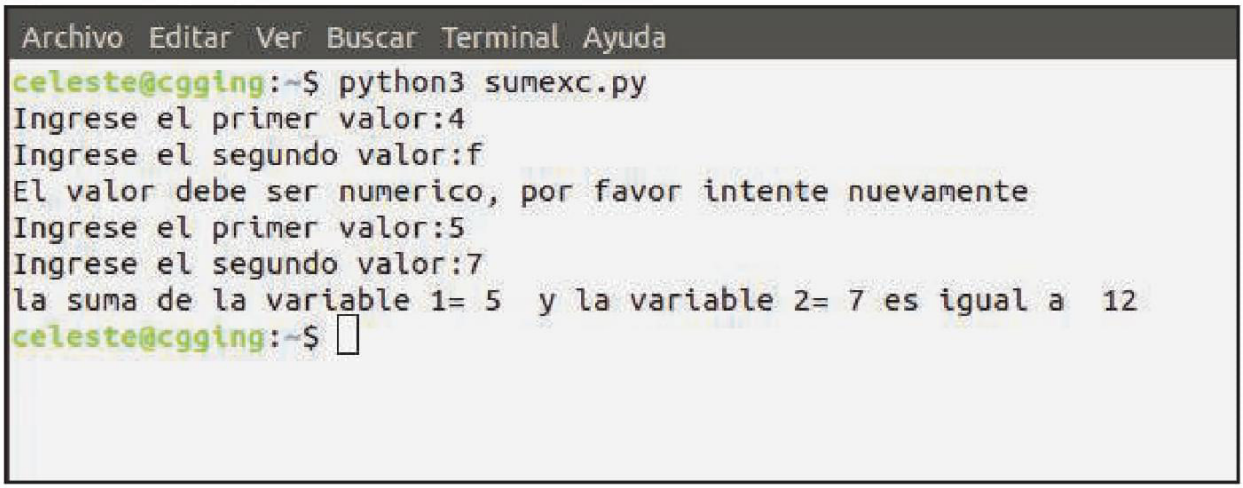
```
while True:
    try:
        variable1=int(input("Ingrese el primer valor:"))
                variable2=int(input("Ingrese el segundo
valor:"))
        break
    exceptValueError:
    print("El valor debe ser numerico, por favor intente
nuevamente")
suma= variable1 + variable2
print("la suma de la variable 1=" , variable1, " y la vari-
able 2=", variable2, "es igual a ", suma)
```

Reacomodamos el código utilizando el mismo **while** para ambas variables y, podemos ver que queda mucho más corto y limpio. Hagamos algunas pruebas y veamos qué diferencias encontramos en su ejecución:

```
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
celeste@cgging:~$ python3 sumexc.py
Ingrese el primer valor:e
El valor debe ser numerico, por favor intente nuevamente
Ingrese el primer valor:4
Ingrese el segundo valor:2
la suma de la variable 1= 4  y la variable 2= 2 es igual a  6
celeste@cgging:~$
```

Como podemos observar en la imagen anterior, si ingresamos incorrectamente el primer valor, nos pide que ingresemos un valor numérico y, al cumplir con el tipo de dato, pasamos al ingreso del segundo valor sin inconvenientes.

Veamos ahora qué pasa si fallamos en el ingreso del segundo valor:



Notamos en la imagen anterior que, si fallamos en el ingreso del segundo valor, tendremos que volver a ingresar ambos valores, ya que una vez ejecutado el bloque **except**, se volverá a ejecutar el código nuevamente desde el comienzo del bloque **try**.

La primera opción que habíamos mostrado maneja el error de cada variable por separado y de esa forma, si nos equivocamos en el ingreso del segundo valor, nos pedirá que reingresemos ese último número sin perder el primero.

En este ejemplo en particular, solo se pide el ingreso de dos valores, por lo que no resulta muy costoso para el usuario realizar el reingreso de los valores, diferente sería el caso si pidiéramos el ingreso de 10 valores al usuario.

OBTENER EL MAYOR

En este ejemplo, nuevamente se nos presenta el problema del ingreso del valor por teclado. Acá, además, podemos mencionar algunos errores comunes de sintaxis, tal como omitir los : luego del **if** o del **else**, o equivocarnos en la indentación:

```
while True:
     try:
          variable1=int(input(“Ingrese el primer valor:”))
         break
     exceptValueError:
```

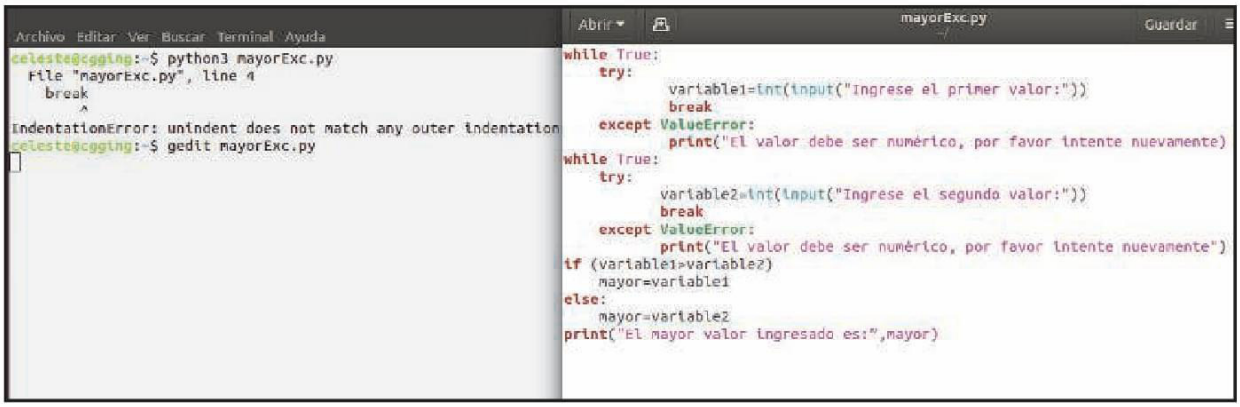
```
     print("El valor debe ser numérico, por favor intente nue-
vamente)
while True:
     try:
         variable2=int(input("Ingrese el segundo valor:"))
        break
     exceptValueError:
     print("El valor debe ser numérico, por favor intente nue-
vamente")
if (variable1>variable2)
    mayor=variable1
else:
    mayor=variable2
print("El mayor valor ingresado es:",mayor)
```

Si intentamos ejecutar el código anterior, pasará lo siguiente:

```
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
celeste@cgging:~$ python3 mayorExc.py
  File "mayorExc.py", line 4
    break
        ^
IndentationError: unindent does not match any outer indentation level
celeste@cgging:~$
```

Vemos que el primer problema con el que nos encontramos es un error de indentación, y también podemos observar que el intérprete nos indica en dónde se está produciendo dicho error. En este caso, como nosotros podemos elegir la cantidad de espacios que utilizaremos para indentar un bloque, el intérprete asume que trabajaremos con la cantidad de espacios que definimos en la declaración de la **`variable1`** y por ese motivo el error se aprecia en **`break`**.

Este error se soluciona editando el archivo y revisando la indentación.



Al volver a ejecutar el script de Python tras corregir el error de indentación, nos encontramos con otro error, esta vez en el **`print`**:

```
Archivo Editar Ver Buscar Terminal Ayuda
celeste@cgging:~$ python3 mayorExc.py
  File "mayorExc.py", line 4
    break
        ^
IndentationError: unindent does not match any outer indentation level
celeste@cgging:~$ gedit mayorExc.py
celeste@cgging:~$ python3 mayorExc.py
  File "mayorExc.py", line 6
    print("El valor debe ser numérico, por favor intente nuevamente)
                                                                     ^
SyntaxError: EOL while scanning string literal
celeste@cgging:~$
```

El intérprete nos indica que se encontró un final de línea antes de que finalizara el texto por imprimir, esto se debe a la omisión de las comillas de cierre del texto.

Al corregir este error y volver a ejecutar el **`script`**, esta vez podemos observar el siguiente error:

Y este error hace referencia a la ausencia de los **`:`** en **`if`**, lo cual deberemos corregir para que nuestro código funcione.

Una vez realizada esta corrección y al intentar nuevamente ejecutar el script, vemos otro error, idéntico al del print que nos apareció antes; no obstante, al mirar el código vemos que las comillas figuran.

```
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
celeste@cgging:~$ python3 mayorExc.py
  File "mayorExc.py", line 4
    break
        ^
IndentationError: unindent does not match any outer indentation level
celeste@cgging:~$ gedit mayorExc.py
celeste@cgging:~$ python3 mayorExc.py
  File "mayorExc.py", line 6
    print("El valor debe ser numérico, por favor intente nuevamente)
                                                                    ^
SyntaxError: EOL while scanning string literal
celeste@cgging:~$ gedit mayorExc.py
celeste@cgging:~$ python3 mayorExc.py
  File "mayorExc.py", line 13
    if (variable1>variable2)
                           ^
SyntaxError: invalid syntax
celeste@cgging:~$
```

Si bien es cierto que tiene comillas de cierre, aquí se produjo el error de copiado que mencionamos al principio del capítulo y podemos ver que las comillas de cierre son diferentes a las comillas de apertura. Para solucionar este error, simplemente cambiamos esas comillas.

```
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
celeste@cgging:~$ python3 mayorExc.py
  File "mayorExc.py", line 4
    break
        ^
IndentationError: unindent does not match any outer indentation level
celeste@cgging:~$ gedit mayorExc.py
celeste@cgging:~$ python3 mayorExc.py
  File "mayorExc.py", line 6
    print("El valor debe ser numérico, por favor intente nuevamente)
                                                                    ^
SyntaxError: EOL while scanning string literal
celeste@cgging:~$ gedit mayorExc.py
celeste@cgging:~$ python3 mayorExc.py
  File "mayorExc.py", line 13
    if (variable1>variable2)
                           ^
SyntaxError: invalid syntax
celeste@cgging:~$ gedit mayorExc.py
celeste@cgging:~$ python3 mayorExc.py
  File "mayorExc.py", line 17
    print("El mayor valor ingresado es:”,mayor)
                                              ^
SyntaxError: EOL while scanning string literal
celeste@cgging:~$ gedit mayorExc.py
```

Abrir mayorExc.py Guardar

```
while True:
    try:
            variable1=int(input("Ingrese el primer valor:"))
            break
    except ValueError:
            print("El valor debe ser numérico, por favor intente nuevamente")
while True:
    try:
           variable2=int(input("Ingrese el segundo valor:"))
           break
    except ValueError:
           print("El valor debe ser numérico, por favor intente nuevamente")
if (variable1>variable2):
    mayor=variable1
else:
    mayor=variable2
print("El mayor valor ingresado es:",mayor)
```

Por último, volvemos a ejecutar el **`script`** con todas las correcciones anteriores:

```
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
celeste@cgging:~$ python3 mayorExc.py
  File "mayorExc.py", line 4
    break
        ^
IndentationError: unindent does not match any outer indentation level
celeste@cgging:~$ gedit mayorExc.py
celeste@cgging:~$ python3 mayorExc.py
  File "mayorExc.py", line 6
    print("El valor debe ser numérico, por favor intente nuevamente)
                                                                    ^
SyntaxError: EOL while scanning string literal
celeste@cgging:~$ gedit mayorExc.py
celeste@cgging:~$ python3 mayorExc.py
  File "mayorExc.py", line 13
    if (variable1>variable2)
                           ^
SyntaxError: invalid syntax
celeste@cgging:~$ gedit mayorExc.py
celeste@cgging:~$ python3 mayorExc.py
  File "mayorExc.py", line 17
    print("El mayor valor ingresado es:",mayor)
                                              ^
SyntaxError: EOL while scanning string literal
celeste@cgging:~$ gedit mayorExc.py
celeste@cgging:~$ python3 mayorExc.py
Ingrese el primer valor:t
El valor debe ser numérico, por favor intente nuevamente
Ingrese el primer valor:3
Ingrese el segundo valor:u
El valor debe ser numérico, por favor intente nuevamente
Ingrese el segundo valor:9
El mayor valor ingresado es: 9
celeste@cgging:~$
```

EJEMPLO FINAL: SUMAR LA DOCUMENTACIÓN DEL CÓDIGO

Como mencionamos varias veces a lo largo de este libro, es importante al programar tener en cuenta ciertas normas implícitas que se consideran buenas rutinas. Una de ellas, que hemos mencionado pero no hemos implementado en los ejemplos anteriores, es la costumbre de documentar o comentar el código que estamos escribiendo. Veamos cómo aplicar esta práctica en el ejemplo integrador que realizamos en el capítulo anterior.

```
fromrandomimport *
#Importamos el modulo random
defgeneraNumeroAleatorio(minimo, maximo):          #definimos
una funcion propia
     returnrandint(minimo, maximo)                 #retornamos
un numero aleatorio

numero_buscado=generaNumeroAleatorio(1,100)        #Llamamos a
la funcion propia
encontrado=False
intentos=0

whilenot encontrado:                                  #Bucle
a repetir mientras negado sea falso
while True:
#Comenzamos el manejo de excepcion
        try:
numero_usuario=int(input("Introduce el numero buscado:  "))
            break
exceptValueError:
#En caso de error de tipo de dato
print("El valor ingresado debe ser un numero, por favor rein-
tente.")
ifnumero_usuario>numero_buscado:                  #compara el
valor random y del usuario
print("El numero que buscas es menor")             #Si el de
```

```
usuario es mayor se ejecuta

# este bloque
        intentos=intentos+1
elifnumero_usuario<numero_buscado:                #Este otro
si el del usuario es menor
print("El numero que buscas es mayor")
        intentos=intentos+1
else:                                                    #Cu-
ando el usuario acierta, se ejecuta este bloque.
        encontrado=True                  #Se cambia el valor
del booleano para terminar el bucle.
print("Has acertado, el numero correcto es ", numero_usuario,
"te ha llevado", intentos, "intentos")
```

Como podemos ver en el ejemplo anterior, se añadieron los comentarios del código que podían aportar a la explicación de su funcionamiento. Esto se considera una buena práctica ya que facilita la tarea de mantenimiento del código e, incluso, nos refresca la memoria en caso de que queramos reutilizarlo para otro proyecto.

Un error común en el momento de documentar el código es agregar información redundante, obvia o inútil, por ejemplo:

```
variable1=25                    #declaramos la variable1
variable2=10                    #declaramos la variable2
suma=variable1+variable2      #sumamos la variable1 y la vari-
able2
print("La suma es", suma)      #mostramos por pantalla la suma
de las variables1 y variable2
```

El código anterior es un claro ejemplo del mal uso de la documentación de un código; podemos ver que tenemos mucha información inútil en los comentarios, que no nos aportan nada más que lo que el propio código nos brinda a simple vista.

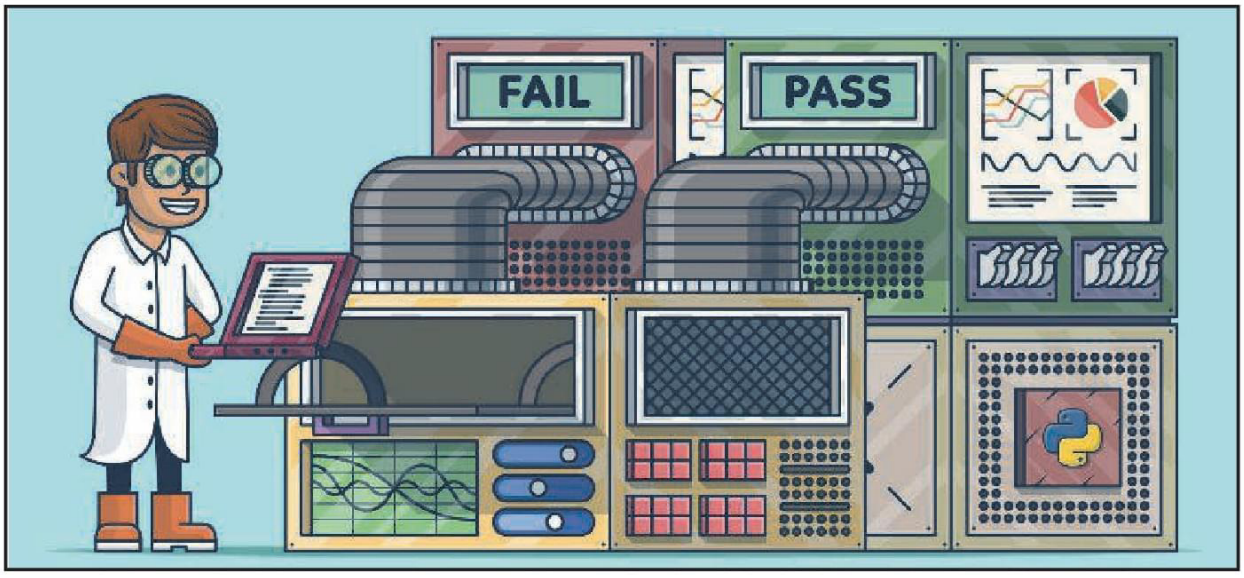
Testear códigos

PRUEBAS AUTOMÁTICAS VERSUS PRUEBAS MANUALES

Es muy probable que ya hayamos realizado testeos sin saberlo y esto es algo positivo. Una de las formas de prueba manual, conocida como **prueba exploratoria**, consiste en ejecutar nuestro código, revisar las características y experimentar como usuario.

Las pruebas exploratorias son una forma de intentos que se realizan sin un plan. En una prueba exploratoria, solo se está sondeando la aplicación.

Para tener un conjunto completo de pruebas manuales, lo que se necesita es confeccionar un listado de todas las características que tiene nuestra aplicación, los diferentes tipos de entrada que se aceptan y los resultados esperados. Ahora, cada vez que realicemos un cambio en nuestro código, tendremos que revisar cada elemento de esa lista y verificarlo. Esto suena bastante engorroso.



Contamos con muchas formas para testear un código. En este capítulo, veremos desde las más básicas hasta algunas avanzadas.

07

Aquí es donde utilizaremos la prueba automatizada. Esta consiste en la ejecución de nuestro plan de prueba (las partes del código que deseamos probar, el orden en el que queremos probarlas y las respuestas esperadas) por un script en lugar de una persona. Python ya viene con un conjunto de herramientas y bibliotecas para ayudarnos a crear pruebas automatizadas para nuestro código. Exploraremos esas herramientas y bibliotecas en este capítulo.

PRUEBAS UNITARIAS VERSUS PRUEBAS DE INTEGRACIÓN

El mundo de las pruebas no tiene escasez de terminología, y ahora que conocemos la diferencia entre las pruebas automáticas y las manuales, es hora de ir más allá.

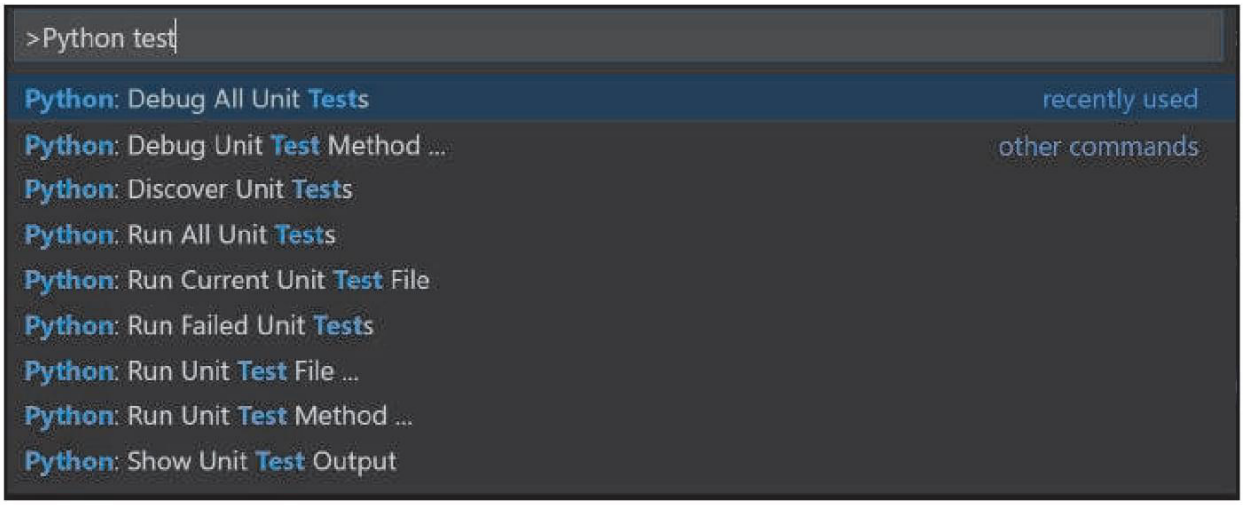
Pensemos en cómo podríamos probar las luces de un auto. Encenderíamos las luces (lo que se conoce como el *paso de prueba*) y saldríamos del auto o pediríamos a alguien que verificara que estuvieran encendidas (lo que se conoce como *afirmación de prueba*). La prueba de múltiples componentes se denomina **prueba de integración**.

Consideremos todas las cosas que deben funcionar correctamente para que una tarea simple dé el resultado correcto. Estos componentes son como las partes de nuestro código, todas esas clases, funciones y módulos que hemos escrito.

Un desafío importante con las pruebas de integración aparece cuando una de ellas no da el resultado correcto. Es muy difícil diagnosticar el problema sin poder aislar qué parte del sistema está fallando. Si las luces no se encienden, entonces quizás las lamparitas estén rotas. ¿La batería está agotada? ¿Qué pasa con el alternador? ¿Está fallando la computadora del auto?

Si tenemos un auto moderno, probablemente nos indique cuando las luces no encienden. Esto se logra usando una forma de **prueba unitaria**.

Una prueba unitaria es una prueba más pequeña, que verifica que un solo componente funciona de la manera correcta. Este tipo de prueba nos ayuda a aislar lo que no funciona en nuestro código y a solucionar más rápido el problema.



Acabamos de ver dos tipos de pruebas: una prueba de integración, que verifica que los componentes de nuestro código operen entre sí, y una prueba unitaria, que verifica un pequeño componente de nuestro código. Podemos escribir tanto pruebas de integración como pruebas unitarias en Python.

PROFUNDIZAR EL CONCEPTO DE UNIT TESTING

Se trata de un método para determinar si un módulo o un conjunto de módulos de código funciona correctamente. El concepto de **Unit Testing** no se limita a ningún lenguaje específico, sino que es una herramienta de la programación en general. Las pruebas unitarias se implementan a la par con el desarrollo de un módulo o proyecto, y se ejecutan cuando este último sufre modificaciones para garantizar su funcionamiento. Si bien el código mismo de la prueba unitaria puede contener errores, la clave está en la separación del código de un módulo de su respectiva prueba unitaria, de modo que puedan correr independientemente.

A simple unit test

```
# test_port1.py

import unittest
from portfolio1 import Portfolio

class PortfolioTest(unittest.TestCase):
    def test_buy_one_stock(self):
        p = Portfolio()
        p.buy("IBM", 100, 176.48)
        assert p.cost() == 17648.0
```

```
$ python -m unittest test_port1
.
----------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

En otras palabras, es una forma de comprobar que un conjunto de funciones o clases (tantas como queramos) actúan como esperamos. Lógicamente, las pruebas unitarias nunca pueden garantizar por completo el correcto funcionamiento de una porción de código. No obstante ello, serán capaces de detectar gran cantidad de anomalías y de ahorrarnos tiempo de depuración.

UTILIZAR UNIT TESTING

Las pruebas unitarias se utilizan siempre que lo consideremos apropiado. Estas resultan una buena práctica a la hora de programar y pueden dar cuenta de un cierto "control de calidad" que garantiza en parte el funcionamiento de nuestro código.

IMPLEMENTAR UNIT TESTING EN PYTHON

Existen diversos frameworks para implementar pruebas unitarias en el lenguaje. En este capítulo trataremos los dos que se incluyen en la librería estándar: **`unittest`** y **`doctest`**. El primero, que ya hemos mencionado en capítulos anteriores, deriva de otros lenguajes y tiene una eficiencia razonable. El segundo está más enfocado a Python. Veremos las características generales de ambos para poder utilizar cualquiera de los dos en nuestros proyectos.

Para comenzar, utilizaremos el siguiente módulo al que llamaremos **mymodule.py** y que, por el momento, tendrá la siguiente función matemática:

```
#!/usr/bin/envpython
# -*- coding: utf-8 -*-

def sum(a, b):
for n in (a, b):
ifnotisinstance(n, int) and notisinstance(n, float):
raiseTypeError
return a + b
```

La función **sum** obtiene dos números (enteros o de coma flotante) y retorna el resultado de su suma. Si se ingresan valores que no sean del tipo comentado antes, se lanza la excepción **TypeError**.

PRUEBA UNITARIA CON UNITTEST

Para la implementación de la prueba unitaria utilizando el módulo comando, crearemos un nuevo archivo llamado **test_mymodule.py** en el mismo directorio que el anterior.

```
#!/usr/bin/envpython
# -*- coding: utf-8 -*-

importunittest
importmymodule


classTestMyModule(unittest.TestCase):

deftest_sum(self):
self.assertEqual(mymodule.sum(5, 7), 12)


if __name__ == “__main__”:
unittest.main()
```

Veamos qué estamos haciendo.

En las primeras dos líneas, importamos el módulo **unittest** necesario para crear las pruebas unitarias y, a continuación, el módulo que queremos probar.

Luego creamos la clase **TestMyModule**, una unidad de prueba que comprobará el comportamiento de nuestro módulo. Dentro de esta creamos tantos métodos como funciones del módulo queremos probar. Todos los métodos que comiencen con el nombre **test** serán ejecutados.

En nuestro caso, queremos comprobar que la función que creamos antes, **mymodule.sum**, funciona correctamente. Para esto, utilizamos una operación básica: 5 + 7 es necesariamente 12. Si el resultado de **mymodule.sum(5, 7)** no es igual a **12** entonces nuestra función contiene algún error, y la prueba unitaria nos lo hará saber.

Al llamar a la función **assertEqual**, estamos indicando que el valor de retorno de **mymodule.sum(5, 7)** debe ser igual (**equal**) a **12**.

La documentación provee una tabla con el resto de las funciones de **unittest** y su respectiva operación.

FUNCIÓN	OPERACIÓN EQUIVALENTE
`assertEqual(a, b)`	a == b
`assertNotEqual(a, b)`	a != b
`assertTrue(x)`	bool(x) is True
`assertFalse(x)`	bool(x) is False
`assertIs(a, b)`	a is b
`assertIsNot(a, b)`	a isnot b
`assertIsNone(x)`	x isNone
`assertIsNotNone(x)`	x isnotNone
`assertIn(a, b)`	a in b
`assertNotIn(a, b)`	a not in b
`assertIsInstance(a, b)`	isinstance(a, b)
`assertNotIsInstance(a, b)`	notisinstance(a, b)

Es conveniente utilizar las funciones de Unit Test en lugar de sus equivalentes, ya que ofrecerán mayor detalle del error cuando el resultado sea diferente al esperado.

Retomando el código de ejemplo, por último llamamos a la función **`unittest.main`** para ejecutar la prueba unitaria. Llamamos a **`python test_mymodule.py`** y el resultado es el siguiente.

```
.
----------------------------------------------------
Ran 1 test in 0.000s

OK
```

El resultado indica que se corrió una prueba y no hubo ningún error. Haremos una pequeña modificación en la función original.

```
def sum(a, b):
for n in (a, b):
ifnotisinstance(n, int) and notisinstance(n, float):
raiseTypeError
return a - b
```

Nótese **`return a - b`**. En este caso, de **`sum(5, 7)`** resultará **`-2`**, mientras que la prueba unitaria espera **`12`**. Volvemos a correrla y obtenemos lo siguiente.

```
F
======================================================
FAIL: test_sum (__main__.TestMyModule)
------------------------------------------------------
Traceback (mostrecentcalllast):
  File “test_mymodule.py”, line 11, in test_sum
self.assertEqual(mymodule.sum(5, 7), 12)
AssertionError: -2 != 12

------------------------------------------------------
```

```
Ran 1 test in 0.000s

FAILED (failures=1)
```

El resultado indica cuántas pruebas fallaron y muestra dónde se produjo el error.

Además de la lista anterior de funciones, también se puede comprobar si nuestro método lanza una excepción en un caso determinado. Por ejemplo, no es posible sumar un número entero con una cadena, según nuestra función, en este caso debería lanzarse **TypeError**. Por consiguiente, podríamos añadir la siguiente comprobación.

```
deftest_sum(self):
self.assertEqual(mymodule.sum(5, 7), 12)
self.assertRaises(TypeError, mymodule.sum, 5, “Python”)
```

El primer argumento de **asserRaises** indica la excepción que se espera; el segundo, la función que queremos probar y, a continuación, sus argumentos. También puede expresarse de la siguiente forma:

```
deftest_sum(self):
self.assertEqual(mymodule.sum(5, 7), 12)
withself.assertRaises(TypeError):
mymodule.sum(5, “Python”)
```

Resulta más agradable y legible.

Para finalizar con esta sección, añadimos a nuestro módulo de funciones matemáticas los siguientes métodos:

```
defget_prime_numbers(max_number):
numbers = [True, True] + [True] * (max_number-1)
last_prime_number = 2
   i = last_prime_number

whilelast_prime_number**2 <= max_number:
```

```
        i += last_prime_number
while i <= max_number:
numbers[i] = False
            i += last_prime_number
        j = last_prime_number + 1
while j <max_number:
ifnumbers[j]:
last_prime_number = j
                break
            j += 1
        i = last_prime_number

return [i + 2 for i, not_crossed in enumerate(numbers[2:])
ifnot_crossed]


defis_prime(n):
return n in get_prime_numbers(n)
```

Y en nuestro archivo de pruebas unitarias incluimos algunas comprobaciones:

```
deftest_get_prime_numbers(self):
        # Los números primos menores a 10 son necesariamente
2, 3, 5 y 7.
self.assertEqual(mymodule.get_prime_numbers(10), [2, 3, 5,
7])

deftest_is_prime(self):
        # 5 es primo.
self.assertTrue(mymodule.is_prime(5))
        # 6 no es primo.
self.assertFalse(mymodule.is_prime(6))
Código completo:
#!/usr/bin/envpython
# -*- coding: utf-8 -*-
```

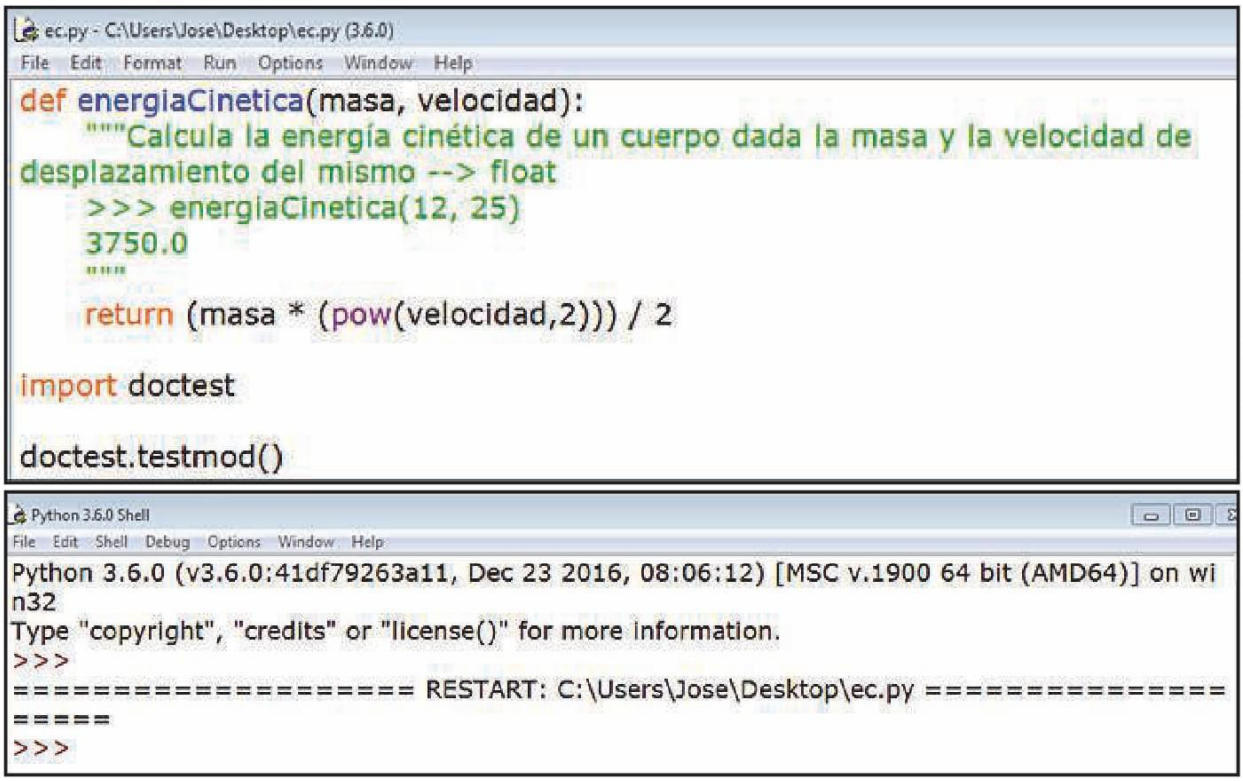
```
importunittest
importmymodule


classTestMyModule(unittest.TestCase):

deftest_get_prime_numbers(self):
self.assertEqual(mymodule.get_prime_numbers(10), [2, 3, 5,
7])

deftest_is_prime(self):
self.assertTrue(mymodule.is_prime(5))
self.assertFalse(mymodule.is_prime(6))

deftest_sum(self):
self.assertEqual(mymodule.sum(5, 7), 12)
withself.assertRaises(TypeError):
mymodule.sum(5, “Python”)


if __name__ == “__main__”:
unittest.main()
```

PRUEBA UNITARIA CON DOCTEST

La implementación de la prueba unitaria utilizando **doctest** se realiza junto con la documentación de una función o clase. Por ejemplo, siguiendo con nuestra función original **sum**, añadiremos una breve descripción en la documentación:

```
def sum(a, b):
    “””
    Retorna a + b.
    “””
for n in (a, b):
```

```
ifnotisinstance(n, int) and notisinstance(n, float):
raiseTypeError
return a + b
```



Debajo de la descripción, agregamos el ejemplo que tratamos:

```
def sum(a, b):
    “””
    Retorna a + b.

>>> sum(5, 7)
    12
    “””
```

doctest ejecuta todas las operaciones que se encuentren luego de **>>>** y las compara con el resultado inmediatamente siguiente hasta otra operación o bien un espacio en blanco.

Esto resulta de gran comodidad ya que la documentación de nuestra función cumple un objetivo dual: ilustrar con un ejemplo y servir como prueba unitaria.

Para ejecutar doctest, añadiremos al final de nuestro módulo **mymodule.py** el siguiente fragmento:

```
if __name__ == “__main__”:
importdoctest
doctest.testmod(verbose=True)
```

De esta forma, cuando el módulo sea ejecutado (no así importado), doctest analizará la documentación de todas las funciones o clases que contiene y ejecutará las pruebas que corresponden. El resultado es el siguiente:

```
Trying:
    sum(5, 7)
Expecting:
    12
ok
1 itemshad no tests:
    __main__
1 itemspassedalltests:
    1 tests in __main__.sum
1 tests in 2 items.
1 passed and 0 failed.
Test passed.
```

Utilizando **verbose=False**, la prueba unitaria imprimirá en pantalla únicamente los mensajes de error.

Este tipo de prueba unitaria puede ser ejecutada desde la terminal sin necesidad de incluir las últimas tres líneas de código que añadimos antes:

```
python -m doctest -v mymodule.py
```

A través de la documentación de nuestra función, también podemos indicar que se espera una excepción cuando intentamos sumar un entero y una cadena:

```
def sum(a, b):
    “””
    Retorna a + b.

>>> sum(5, 7)
    12
>>> sum(5, “Python”)
Traceback (mostrecentcalllast):
        ...
TypeError
    “””
```

Dado que los mensajes de error contienen información que varía constantemente (nombre del archivo, número de línea, etcétera), no es necesario indicarlo y, en su lugar, se colocan tres puntos.

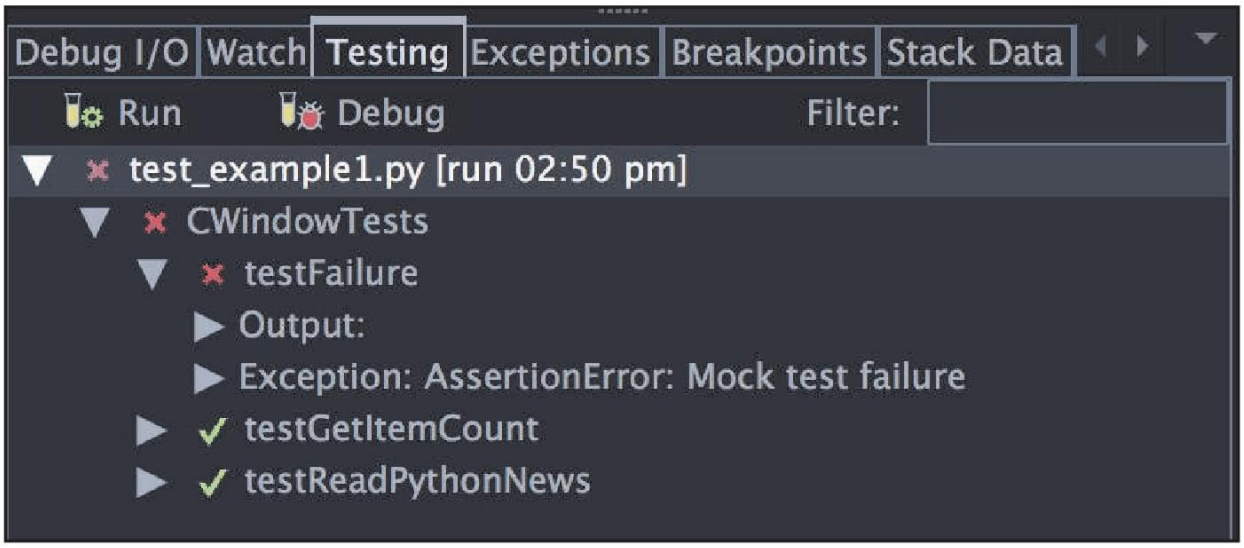
Para finalizar completamos las otras dos funciones con su respectiva documentación y ejemplos:

```
defget_prime_numbers(max_number):
    “””
    Retorna una lista de números primos.

>>>get_prime_numbers(10)
    [2, 3, 5, 7]
    “””
defis_prime(n):
    “””
    Determina si n es primo.

>>>is_prime(5)
    True
>>>is_prime(6)
    False
    “””
```

Código completo:

```
#!/usr/bin/envpython
# -*- coding: utf-8 -*-

defget_prime_numbers(max_number):
    “””
    Retorna una lista de números primos.

>>>get_prime_numbers(10)
    [2, 3, 5, 7]
    “””
numbers = [True, True] + [True] * (max_number-1)
last_prime_number = 2
    i = last_prime_number

whilelast_prime_number**2 <= max_number:
        i += last_prime_number
while i <= max_number:
numbers[i] = False
            i += last_prime_number
        j = last_prime_number + 1
while j <max_number:
ifnumbers[j]:
last_prime_number = j
                break
            j += 1
        i = last_prime_number

return [i + 2 for i, not_crossed in enumerate(numbers[2:])
ifnot_crossed]


defis_prime(n):
    “””
    Determina si n es primo.

>>>is_prime(5)
    True
>>>is_prime(6)
```

```
   False
   “””
return n in get_prime_numbers(n)


def sum(a, b):
   “””
   Retorna a + b.

>>> sum(5, 7)
   12
>>> sum(5, “Python”)
Traceback (mostrecentcalllast):
      ...
TypeError
   “””
for n in (a, b):
ifnotisinstance(n, int) and notisinstance(n, float):
raiseTypeError
return a + b


if __name__ == “__main__”:
importdoctest
doctest.testmod(verbose=False)
```



RedUSERS PREMIUM

- Nuestros expertos, a través de cientos de publicaciones, comparten su saber y experiencia
- Calidad y cantidad por una mínima cuota mensual
- Con los temas que más interesan
- Siempre, donde vayas. On Line - Off Line. En cualquier dispositivo
- Al menos 1 novedad semanal. Lee todo lo que desees sin límites. Son 700 publicaciones y sumando....!!

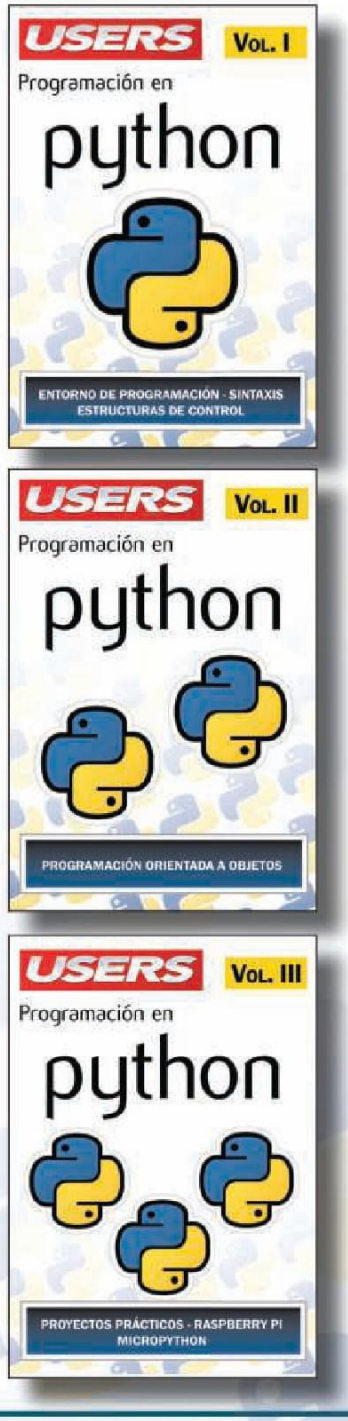
SUSCRÍBETE

usershop.redusers.com
+54-11-4110-8700
usershop@redusers.com

USERS

VOL. II

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 II

En el segundo volumen de los tres que componen la serie, presentamos el paradigma de programación orientada a objetos, revisamos las clases, herencia y todo el campo de posibilidades que nos abre comenzar a utilizarlo en Python.

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.

REDUSERS.PREMIUM.COM

RedUSERS Premium es la biblioteca digital de USERS. Accederás a cientos de publicaciones: Informes; eBooks; Guías; Revistas; Cursos. Todo el contenido está disponible Online-Offline y para cualquier dispositivo. Publicamos, al menos, una novedad cada 7 días

ISBN 978-987-4958-19-8

REDUSERS.COM

En nuestro sitio podrá encontrar noticias relacionadas y participar de la comunidad de tecnología más importante de América Latina.