Python Frequently Asked Questions

Versión 3.8.20

Guido van Rossum
and the Python development team

septiembre 08, 2024

Python Software Foundation
Email: docs@python.org

Índice general

CAPÍTULO 1

Preguntas frecuentes generales sobre Python

1.1 Información General

1.1.1 ¿Qué es Python?

Python is an interpreted, interactive, object-oriented programming language. It incorporates modules, exceptions, dynamic typing, very high level dynamic data types, and classes. It supports multiple programming paradigms beyond object-oriented programming, such as procedural and functional programming. Python combines remarkable power with very clear syntax. It has interfaces to many system calls and libraries, as well as to various window systems, and is extensible in C or C++. It is also usable as an extension language for applications that need a programmable interface. Finally, Python is portable: it runs on many Unix variants including Linux and macOS, and on Windows.

Para saber más, comienza con tutorial-index. La Beginner's Guide to Python vincula a otros recursos y tutoriales introductorios para aprender Python.

1.1.2 ¿Que es la *Python Software Foundation*?

La *Python Software Foundation* es una organización independiente sin fines de lucro que posee los derechos sobre Python desde la versión 2.1 en adelante. La misión de la PSF es hacer avanzar la tecnología *open source* relacionada al lenguaje de programación Python y publicitar su uso. El sitio web de la PSF es https://www.python.org/psf/.

Las donaciones a la PSF están exentas de impuestos en Estados Unidos. Si usas Python y lo encuentras útil, por favor contribuye a través de la página de donaciones de la PSF.

1.1.3 ¿Hay restricciones de *copyright* sobre el uso de Python?

Puedes hacer cualquier cosa que quieras con el código fuente mientras mantengas y muestres los mensajes de *copyrights* en cualquier documentación sobre Python que produzcas. Si respetas las reglas de *copyright*, está permitido usar Python para fines comerciales, vender copias de Python en forma de código fuente o binarios (modificados o no), o vender productos que incorporen Python de alguna manera. De cualquier manera nos gustaría saber de todos los usos comerciales de Python, por supuesto.

Mira la página PSF license para encontrar explicaciones más detalladas y un vínculo al texto completo de la licencia.

El logo de Python tiene derechos comerciales (*trademarked*) y en ciertos casos se requiere un permiso de uso. Consulta la Trademark Usage Policy para más información.

1.1.4 ¿Por qué motivos se creó Python?

Aquí hay un *muy* breve resumen sobre qué fue lo que comenzó todo, escrita por Guido van Rossum:

> Tenía vasta experiencia implementando un lenguaje interpretado en el grupo ABC en CWI y trabajando con este grupo había aprendido mucho sobre diseño de lenguajes. Este es el origen de muchas características de Python, incluyendo el uso de sangría para el agrupamiento de sentencias y la inclusión de tipos de datos de muy alto nivel (aunque los detalles son todos diferentes en Python).
>
> Tenía algunos resquemores sobre el lenguaje ABC pero también me gustaban muchas de sus características. Era imposible extenderlo (al lenguaje o sus implementaciones) para remediar mis quejas – de hecho, la ausencia de extensibilidad fue uno de los mayores problemas. Contaba con alguna experiencia usando Modula-2+ y conversé con los diseñadores de Modula-3 y leí su reporte. Modula-3 es el origen de la sintaxis y semántica que usé para las excepciones y otras características de Python.
>
> Estaba trabajando en Grupo del sistema operativo distribuido Amoeba en CWI. Necesitábamos una mejor manera de hacer administración de sistemas que escribir programas en C o *scripts* de *Bourne shell*, ya que Amoeba tenía sus propia interfaz de llamadas a sistema que no era fácilmente accesible desde *Bourne shell*. Mi experiencia con el manejo de errores de Amoeba me hizo muy consciente de la importancia de las excepciones como una característica de los lenguaje de programación.
>
> Se me ocurrió que un lenguaje de scripts con una sintaxis como ABC pero con acceso a las llamadas al sistema Amoeba satisfaría la necesidad. Me di cuenta de que sería una tontería escribir un lenguaje específico para Amoeba, así que decidí que necesitaba un lenguaje que fuera generalmente extensible.
>
> Durante las vacaciones de Navidad de 1989 tenía mucho tiempo libre, así que decidí hacer un intento. Durante el año siguiente, mientras seguía trabajando principalmente en él durante mi propio tiempo, Python se utilizó en el proyecto Amoeba con un éxito creciente, y los comentarios de mis colegas me hicieron agregar muchas mejoras iniciales.
>
> En febrero de 1991, justo después de un año de desarrollo, decidí publicarlo en USENET. El resto está en el archivo `Misc/HISTORY`.

1.1.5 ¿Para qué es bueno Python?

Python es un lenguaje de programación de propósito general de alto nivel que se puede aplicar a muchas clases diferentes de problemas.

El lenguaje viene con una vasta biblioteca estándar que cubre áreas como el procesamiento de texto (expresiones regulares, unicode, cálculo de diferencias entre archivos), protocolos de Internet (HTTP, FTP, SMTP, XML-RPC, POP, IMAP, programación CGI), ingeniería de software (pruebas unitarias, *logging*, perfilamiento, análisis sintáctico y gramatical de código Python) e interfaces con el sistema operativo (llamadas a sistema, sistemas de archivo, *sockets* TCP/IP). Mira la tabla de contenidos en library-index para tener una idea de qué está disponible. Una amplia variedad de extensiones de terceros también están disponibles. Consulta el Python Package Index para encontrar paquetes de tu interés.

1.1.6 ¿Cómo funciona el esquema numérico de versiones de Python?

La versiones de Python están numeradas A.B.C o A.B. A es el numero de versión más importante – sólo es incrementado por cambios realmente grandes en el lenguaje. B es el número de versión secundario (o menor), incrementado ante cambios menos traumáticos. C es el nivel micro – se incrementa en cada lanzamiento de corrección de errores. Mira el **PEP 6** para más información sobre los lanzamientos de corrección de errores.

No todos los lanzamientos son de corrección de errores. En el período previo a un lanzamiento importante, una serie de lanzamientos de desarrollo son realizados, denotados como *alpha*, *beta* o *release candidate*. Las versiones *alphas* son lanzamientos tempranos en los que las interfaces no están todavía finalizadas; no es inesperado que una interfaz cambie entre dos lanzamientos *alpha*. Las *betas* son más estables, preservando las interfaces existentes pero posiblemente agregando nuevos módulos. Los *release candidates* están congelados, sin hacer cambios excepto los necesarios para corregir bugs críticos.

Las versiones *alpha*, *beta* y *release candidates* tienen un sufijo adicional. El sufijo para la versión alpha es «aN» para algunos números N pequeños; el sufijo para beta es «bN» para algunos números N pequeños, y el sufijo para *release candidates* es «cN» para algunos números N pequeños. En otras palabras, todas las versiones etiquetadas 2.0aN preceden a las 2.0bN, que preceden a las etiquetadas 2.0cN, y *todas esas* preceden a la 2.0.

También puedes encontrar números de versión con un sufijo «+», por ejemplo «2.2+». Estas son versiones sin lanzar, construidas directamente desde el repositorio de desarrollo de CPython. En la práctica, luego de que un lanzamiento menor se realiza, la versión es incrementada a la siguiente versión menor, que se vuelve «a0», por ejemplo «2.4a0».

Mira también la documentación para `sys.version`, `sys.hexversion`, y `sys.version_info`.

1.1.7 ¿Cómo obtengo una copia del código fuente de Python?

El código fuente de la versión más reciente de Python está siempre disponible desde python.org, en https://www.python.org/downloads/. El código fuente en desarrollo más reciente se puede obtener en https://github.com/python/cpython/.

La distribución de fuentes es un archivo tar comprimido con gzip que contiene el código C completo, documentación en formato Sphinx, los módulos de la biblioteca de Python, programas de ejemplo y varias piezas útiles de software libremente distribuibles. El código fuente compilará y se ejecutará sin problemas en la mayoría de las plataformas Unix.

Consulta Getting Started section of the Python Developer's Guide para más información sobre cómo obtener el código fuente y compilarlo.

1.1.8 ¿Cómo consigo documentación sobre Python?

La documentación estándar para la versión estable actual de Python está disponible en https://docs.python.org/3/. También están disponibles versiones en PDF, texto plano y HTML descargable en https://docs.python.org/3/download.html.

La documentación está escrita en reStructuredText y procesada con la herramienta de documentación Sphinx. Las fuentes reStructuredText de la documentación son parte de la distribución fuente de Python.

1.1.9 Nunca he programado antes. ¿Hay un tutorial de Python?

Hay numerosos tutoriales y libros disponibles. La documentación estándar incluye tutorial-index.

Consulta the Beginner's Guide para encontrar información para principiantes en Python, incluyendo una lista de tutoriales.

1.1.10 ¿Hay un *newsgroup* o una lista de correo dedicada a Python?

Hay un grupo de noticias, `comp.lang.python`, y una lista de correo, python-list. Tanto el grupo de noticias como la lista de correo están interconectadas entre sí – si puedes leer las noticias no es necesario que te suscribas a la lista de correo. `comp.lang.python` tiene mucho tráfico, recibiendo cientos de publicaciones cada día. y los lectores de Usenet suelen ser más capaces de hacer frente a este volumen.

Los anuncios de nuevos lanzamientos de software y eventos se pueden encontrar en comp.lang.python.announce, una lista moderada de bajo tráfico que recibe alrededor de cinco publicaciones por día. Está disponible como la lista de correos de anuncios de Python.

Más información sobre listas de correo o grupos de noticias puede hallarse en https://www.python.org/community/lists/.

1.1.11 ¿Cómo obtengo una versión de prueba *beta* de Python?

Las versiones alpha y beta están disponibles desde https://www.python.org/downloads/. Todos los lanzamientos son anunciados en el grupo de noticias comp.lang.python y comp.lang.python.announce, así como también en la página principal de Python en https://www.python.org/; un *feed* RSS está disponible.

También puedes acceder a la versión en desarrollo de Python desde Git. Mira The Python Developer's Guide para los detalles.

1.1.12 ¿Cómo envío un reporte de *bug* y parches para Python?

Para reportar un *bug* o enviar un parche, por favor usa la instalación de Roundup en https://bugs.python.org/.

Debes tener una cuenta de Roundup para reportar *bugs*; esto nos permite contactarte si tenemos más preguntas. También permite que Roundup te envíe actualizaciones cuando haya actualizaciones sobre tu *bug*. Si previamente usaste SourceForge para reportar bugs a Python, puedes obtener tu contraseña de Roundup a través del procedimiento de reinicio de contraseña de Roundup.

Para más información sobre cómo se desarrolla Python, consulta the Python Developer's Guide.

1.1.13 ¿Hay algún artículo publicado sobre Python que pueda referir?

Lo más probable es que lo mejor sea citar a tu libro preferido sobre Python.

El primer artículo publicado sobre Python fue escrito en 1991 y quedó bastante desactualizado.

> Guido van Rossum y Jelke de Boer, «*Interactively Testing Remote Servers Using the Python Programming Language», CWI Quarterly, Volume 4, Issue 4 (Diembre de 1991), Amsterdam, pp 283–303*.

1.1.14 ¿Hay libros sobre Python?

Sí, hay muchos, y hay más siendo publicados. Mira la wiki de python.org en https://wiki.python.org/moin/PythonBooks para ver una lista.

También puedes buscar «Python» en las librerías online y excluir las que refieran a los Monty Python; o quizás buscar «Python» y «lenguaje».

1.1.15 ¿En qué parte del mundo está ubicado www.python.org?

La infraestructura del proyecto Python está ubicada alrededor de todo el mundo y es gestionada por el *Python Infrastructure Team*. Detalles aquí.

1.1.16 ¿Por qué se llama Python?

Cuando comenzó a implementar Python, Guido van Rossum también estaba leyendo los guiones publicados de «Monty Python's Flying Circus», una serie de comedia producida por la BBC de los 70". Van Rossum pensó que necesitaba un nombre que fuera corto, único y ligeramente misterioso, entonces decidió llamar al lenguaje Python.

1.1.17 ¿Debe gustarme «Monty Python's Flying Circus»?

No, pero ayuda. :)

1.2 Python en el mundo real

1.2.1 ¿Cuán estable es Python?

Very stable. New, stable releases have been coming out roughly every 6 to 18 months since 1991, and this seems likely to continue. As of version 3.9, Python will have a major new release every 12 months (**PEP 602**).

Los desarrolladores publican lanzamientos de «bugfix» (corrección de errores) para versiones antiguas, así que la estabilidad de las versiones existentes mejora gradualmente. Los lanzamientos de corrección de errores, indicados por el tercer componente del número de versión (e.g 3.5.3, 3.6.2) son gestionados para estabilidad; sólo correcciones de problemas conocidos se incluyen en uno de estos lanzamientos, y está garantizado que las interfaces se mantendrán a lo largo de la misma serie.

La última versión estable siempre se puede encontrar en la página Python download page. Hay dos versiones de Python que están listas para producción: la 3.x y la 2.x. La versión recomendada es la 3.x, que es soportada por la mayoría de las bibliotecas más usadas. Aunque la versión 2.x aún se usa, no es mantenida desde el 1º de enero de 2020.

1.2.2 ¿Cuánta gente usa Python?

There are probably millions of users, though it's difficult to obtain an exact count.

Python está disponible gratuitamente para ser descargado por lo que no existen cifras de ventas, a su vez se incluye en muchos sitios diferentes y está empaquetado en muchas distribuciones de Linux, por lo que las estadísticas de descarga tampoco cuentan toda la historia.

El grupo de noticias comp.lang.python es muy activo, pero no todos los usuarios de Python publican allí o incluso lo leen.

1.2.3 ¿Hay proyectos significativos hechos con Python?

Mira https://www.python.org/about/success para una lista de proyecto que usan Python. Consultar las actas de conferencias de Python pasadas revelará contribuciones de diferentes empresas y organizaciones.

Proyectos en Python de alto perfil incluyen el gestor de listas de correo Mailman y el servidor de aplicaciones Zope. Muchas distribuciones de Linux, más notoriamente Red Hat, han escrito partes de sus instaladores y software de administración en Python. Entre las empresas que usan Python internamente se encuentran Google, Yahoo y Lucasfilm Ltd.

1.2.4 ¿Qué nuevos desarrollos se esperan para Python en el futuro?

Mira https://www.python.org/dev/peps/ para las *Python Enhancement Proposals* («Propuestas de mejora de Python», PEPs). Las PEPs son documentos de diseño que describen una nueva funcionalidad sugerida para Python, proveyendo una especificación técnica concisa y una razón fundamental. Busca una PEP titulada «Python X.Y Release Schedule», donde X.Y es una versión que aún no ha sido publicada.

Los nuevos desarrollos son discutidos en la lista de correo python-dev.

1.2.5 ¿Es razonable proponer cambios incompatibles a Python?

En general no. Ya existen millones de líneas de código Python alrededor del mundo, por lo que cualquier cambio en el lenguaje que invalide más que una fracción muy pequeña de los programas existentes tiene que ser mal visto. Incluso si puedes proporcionar un programa de conversión, todavía existe el problema de actualizar toda la documentación; se han escrito muchos libros sobre Python y no queremos invalidarlos a todos de un plumazo.

Si una funcionalidad se debe cambiar, es necesario proporcionar una ruta de actualización gradual. **PEP 5** describe el procedimiento seguido para introducir cambios incompatibles con versiones anteriores para minimizar disrupciones a los usuarios y usuarias.

1.2.6 ¿Python es un buen lenguaje para principiantes?

Sí.

Todavía es común hacer comenzar a estudiantes con lenguajes procedimentales de tipado estático como Pascal, C o un subconjunto de C++ o Java. Los y las estudiantes pueden verse favorecidos si aprenden Python como primer lenguaje. Python tiene una sintaxis simple y consistente y una gran biblioteca estándar. Y, más importante, usar Python en cursos introductorios de programación permite a los estudiantes concentrarse en lo importante de las habilidades de programación como la descomposición de problemas y el diseño de tipos de datos. Con Python los estudiantes pueden ser rápidamente introducidos a conceptos como bucles y procedimientos. Incluso puede trabajar con objetos definidos por el usuario en su primer curso.

Para estudiantes que nunca han programado antes, usar un lenguaje de tipado estático parece antinatural. Presenta complejidades adicionales que deben ser dominadas y ralentizan el ritmo del curso. Quienes están aprendiendo intentan pensar como la computadora, descomponer problemas, diseñar interfaces consistentes y encapsular datos. Si bien aprender a usar un lenguaje de tipado estático es importante en el largo plazo, no es necesariamente el mejor tema a tratar en un primer curso de programación.

Muchos otros aspectos de Python lo vuelven un buen primer lenguaje. Como Java, Python tiene una biblioteca estándar, de manera que los y las estudiantes pueden recibir, de manera temprana, consignas para realizar proyectos de programación que *hagan* algo. Estas consignas no están restringidas a las típicas calculadoras de cuatro operaciones o programas de balances contables. Al usar la biblioteca estándar, pueden ganar la satisfacción de trabajar en aplicaciones realistas mientras aprenden los fundamentos de la programación. Usar la biblioteca estándar también enseña a reusar código. Módulos de terceros, como PyGame, también ayudan a extender los alcances de los y las estudiantes.

El intérprete interactivo de Python les permite probar funcionalidades del lenguaje mientras programan. Pueden tener una ventana con el intérprete corriendo mientras escriben el código de su programa en otra. Si no recuerdan los métodos para una lista, pueden hacer algo así:

```
>>> L = []
>>> dir(L)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
'__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__getitem__', '__gt__', '__hash__', '__iadd__',
'__imul__', '__init__', '__iter__', '__le__', '__len__', '__lt__',
'__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__',
'__sizeof__', '__str__', '__subclasshook__', 'append', 'clear',
'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove',
'reverse', 'sort']
>>> [d for d in dir(L) if '__' not in d]
['append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove',
↪'reverse', 'sort']

>>> help(L.append)
Help on built-in function append:

append(...)
    L.append(object) -> None -- append object to end
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> L.append(1)
>>> L
[1]
```

Con el intérprete, la documentación nunca está lejos de los o las estudiantes mientras están programando.

También hay buenas IDEs para Python. IDLE es una IDE multiplataforma para Python que está escrita en Python usando Tkinter. PythonWin es un IDE específico para Windows. Quienes usan Emacs estarán felices de saber que hay un modo para Python muy bueno. Todos estos entornos de programación proveen resaltado de sintaxis, auto-sangrado y acceso al intérprete interactivo mientras se programa. Consulta la wiki de Python para ver una lista completa de entornos de programación.

Si quieres discutir el uso de Python en la educación, quizás te interese unirte a la la lista de correo edu-sig.

CAPÍTULO 2

Preguntas frecuentes de programación

2.1 Preguntas generales

2.1.1 ¿Existe un depurador a nivel de código fuente con puntos de interrupción, depuración paso a paso, etc?

Sí.

Debajo se describen algunos depuradores para Python y la función integrada `breakpoint()` te permite ejecutar alguno de ellos.

El módulo pdb es en depurador en modo consola simple pero conveniente para Python. Es parte de la biblioteca estándar de Python y está `documentado en el manual de referencia de la biblioteca`. Puedes escribir tu propio depurador usando el código de pdb como ejemplo.

El entorno interactivo de desarrollo IDLE, el cual es parte de la distribución Python estándar (disponible, generalmente, como Tools/scripts/idle), incluye un depurador gráfico.

PythonWin is a Python IDE that includes a GUI debugger based on pdb. The PythonWin debugger colors breakpoints and has quite a few cool features such as debugging non-PythonWin programs. PythonWin is available as part of pywin32 project and as a part of the ActivePython distribution.

Eric es un IDE creado usando PyQt y el componente de edición Scintilla.

trepan3k is a gdb-like debugger.

Visual Studio Code is an IDE with debugging tools that integrates with version-control software.

Existen varios IDEs comerciales para Python que incluyen depuradores gráficos. Entre ellos tenemos:

- Wing IDE
- Komodo IDE
- PyCharm

2.1.2 Are there tools to help find bugs or perform static analysis?

Sí.

Pylint and Pyflakes do basic checking that will help you catch bugs sooner.

Inspectores estáticos de tipos como Mypy, Pyre, y Pytype pueden hacer comprobaciones de las anotaciones de tipos en código fuente Python.

2.1.3 ¿Cómo puedo crear un binario independiente a partir de un programa Python?

No necesitas tener la habilidad de compilar Python a código C si lo único que necesitas es un programa independiente que los usuarios puedan descargar y ejecutar sin necesidad de instalar primero una distribución Python. Existe una serie de herramientas que determinan el conjunto de módulos que necesita un programa y une estos módulos conjuntamente con un binario Python para generar un único ejecutable.

Una forma es usando la herramienta *freeze*, la cual viene incluida con el árbol de código Python como `Tools/freeze`. Convierte el byte code Python a arrays C; un compilador C permite incrustar todos tus módulos en un nuevo programa que, posteriormente se puede enlazar con los módulos estándar de Python.

Funciona escaneando su fuente de forma recursiva en busca de declaraciones de importación (en ambas formas) y buscando los módulos en la ruta estándar de Python, así como en el directorio de la fuente (para los módulos incorporados). Luego convierte el *bytecode* de los módulos escritos en Python en código C (inicializadores de arrays que pueden ser convertidos en objetos de código usando el módulo marshal) y crea un archivo de configuración a medida que sólo contiene aquellos módulos incorporados que se usan realmente en el programa. A continuación, compila el código C generado y lo enlaza con el resto del intérprete de Python para formar un binario autónomo que actúa exactamente igual que su script.

Obviously, freeze requires a C compiler. There are several other utilities which don't:

- py2exe for Windows binaries
- py2app for Mac OS X binaries
- cx_Freeze for cross-platform binaries

2.1.4 ¿Existen estándares de código o una guía de estilo para programas Python?

Sí. El estilo de código requerido para los módulos de la biblioteca estándar se encuentra documentado como **PEP 8**.

2.2 Núcleo del lenguaje

2.2.1 ¿Por qué obtengo un *UnboundLocalError* cuando la variable tiene un valor?

Puede ser una sorpresa el hecho de obtener un UnboundLocalError en código que había estado funcionando previamente cuando se modifica mediante el añadido de una declaración de asignación en alguna parte del cuerpo de una función.

Este código:

```
>>> x = 10
>>> def bar():
...     print(x)
>>> bar()
10
```

funciona, pero este código:

```
>>> x = 10
>>> def foo():
...     print(x)
...     x += 1
```

resulta en un UnboundLocalError:

```
>>> foo()
Traceback (most recent call last):
  ...
UnboundLocalError: local variable 'x' referenced before assignment
```

Esto es debido a que cuando realizas una asignación a una variable en un ámbito de aplicación, esa variable se convierte en local y enmascara cualquier variable llamada de forma similar en un ámbito de aplicación exterior. Desde la última declaración en foo asigna un nuevo valor a `x`, el compilador la reconoce como una variable local. Consecuentemente, cuando el `print(x)` más próximo intenta mostrar la variable local no inicializada se muestra un error.

En el ejemplo anterior puedes acceder al ámbito de aplicación exterior a la variable declarándola como global:

```
>>> x = 10
>>> def foobar():
...     global x
...     print(x)
...     x += 1
>>> foobar()
10
```

Esta declaración explícita es necesaria de cara a recordarte que (a diferencia de la situación superficialmente análoga con las variables de clase e instancia) estás modificando el valor de la variable en un ámbito de aplicación más externo:

```
>>> print(x)
11
```

Puedes hacer algo similar en un ámbito de aplicación anidado usando la palabra clave `nonlocal`:

```
>>> def foo():
...     x = 10
...     def bar():
...         nonlocal x
...         print(x)
...         x += 1
...     bar()
...     print(x)
>>> foo()
10
11
```

2.2.2 ¿Cuáles son las reglas para las variables locales y globales en Python?

En Python, las variables que solo se encuentran referenciadas dentro de una función son globales implícitamente. Si a una variable se le asigna un valor en cualquier lugar dentro del cuerpo de una función, se asumirá que es local a no ser que explícitamente se la declare como global.

Aunque, inicialmente, puede parecer sorprendente, un momento de consideración permite explicar esto. Por una parte, requerir `global` para variables asignadas proporciona una barrera frente a efectos secundarios indeseados. Por otra parte, si `global` es requerido para todas las referencias globales, deberás usar `global` en todo momento. Deberías declarar como global cualquier referencia a una función integrada o a un componente de un módulo importado. Este embrollo arruinaría la utilidad de la declaración «global» para identificar los efectos secundarios.

2.2.3 ¿Por qué las funciones lambda definidas en un bucle con diferentes valores devuelven todas el mismo resultado?

Considera que usas un bucle *for* para crear unas pocas funciones lambda (o, incluso, funciones normales), por ejemplo.:

```
>>> squares = []
>>> for x in range(5):
...     squares.append(lambda: x**2)
```

Lo siguiente proporciona una lista que contiene 5 funciones lambda que calculan `x**2`. Esperarías que, cuando se les invoca, retornaran, respectivamente, `0`, `1`, `4`, `9``y ``16`. Sin embargo, cuando lo ejecutes verás que todas devuelven `16`:

```
>>> squares[2]()
16
>>> squares[4]()
16
```

Esto sucede porque `x` no es una función lambda local pero se encuentra definida en un ámbito de aplicación externo y se accede cuando la lambda es invocada — no cuando ha sido definida. Al final del bucle, el valor de `x` es `4`, por tanto, ahora todas las funciones devuelven `4**2`, i.e. `16`. También puedes verificar esto mediante el cambio del valor de `x` y ver como los resultados de las lambdas cambian:

```
>>> x = 8
>>> squares[2]()
64
```

De cara a evitar esto necesitas guardar los valores en variables locales a las funciones lambda de tal forma que no dependan del valor de la `x` global:

```
>>> squares = []
>>> for x in range(5):
...     squares.append(lambda n=x: n**2)
```

Aquí, `n=x` crea una nueva variable `n` local a la función lambda y ejecutada cuando la función lambda se define de tal forma que tiene el mismo valor que tenía `x` en ese punto en el bucle. Esto significa que el valor de `n` será `0` en la primera función lambda, `1` en la segunda, `2` en la tercera y así sucesivamente. Por tanto, ahora cada lambda retornará el resultado correcto:

```
>>> squares[2]()
4
>>> squares[4]()
16
```

Es de destacar que este comportamiento no es peculiar de las funciones lambda sino que aplica también a las funciones regulares.

2.2.4 ¿Cómo puedo compartir variables globales entre módulos?

La forma canónica de compartir información entre módulos dentro de un mismo programa sería creando un módulo especial (a menudo llamado config o cfg). Simplemente importa el módulo config en todos los módulos de tu aplicación; el módulo estará disponible como un nombre global. Debido a que solo hay una instancia de cada módulo, cualquier cambio hecho en el objeto módulo se reflejará en todos los sitios. Por ejemplo:

config.py:

```
x = 0   # Default value of the 'x' configuration setting
```

mod.py:

```
import config
config.x = 1
```

main.py:

```
import config
import mod
print(config.x)
```

Ten en cuenta que usar un módulo es también la base para la implementación del patrón de diseño Singleton, por la misma razón.

2.2.5 ¿Cuáles son las «buenas prácticas» para usar import en un módulo?

En general, no uses `from modulename import *`. Haciendo eso embarulla el espacio de nombres del importador y hace que sea más difícil para los *linters* el detectar los nombres sin definir.

Importar los módulos en la parte inicial del fichero. Haciéndolo así deja claro los módulos que son necesarios para tu código y evita preguntas sobre si el nombre del módulo se encuentra en el ámbito de la aplicación. Usar una importación por línea hace que sea sencillo añadir y eliminar módulos importados pero usar múltiples importaciones por línea usa menos espacio de pantalla.

Es una buena práctica si importas los módulos en el orden siguiente:

1. módulos de la biblioteca estándar – por ejemplo, `sys`, `os`, `getopt`, `re`
2. módulos de bibliotecas de terceros (cualquier cosa instalada en el directorio *site-packages* de Python) – por ejemplo, mx.DateTime, ZODB, PIL.Image, etc.
3. módulos desarrollados localmente

Hay veces en que es necesario mover las importaciones a una función o clase para evitar problemas de importaciones circulares. Gordon McMillan dice:

> No hay problema con las importaciones circulares cuando ambos módulos usan la forma de importación «import <module>». Fallará cuando el segundo módulo quiera coger un nombre del primer módulo («from module import name») y la importación se encuentre en el nivel superior. Esto sucede porque los nombres en el primero todavía no se encuentran disponibles debido a que el primer módulo se encuentra ocupado importando al segundo.

En este caso, si el segundo módulo se usa solamente desde una función, la importación se puede mover de forma sencilla dentro de la función. En el momento en que se invoca a la importación el primer módulo habrá terminado de inicializarse y el segundo módulo podrá hacer la importación.

También podría ser necesario mover importaciones fuera del nivel superior del código si alguno de loa módulos son específicos a la plataforma. En ese caso podría, incluso, no ser posible importar todos los módulos en la parte superior del fichero. Para esos casos, la importación correcta de los módulos en el código correspondiente específico de la plataforma es una buena opción.

Solo debes mover importaciones a un ámbito de aplicación local, como dentro de la definición de una función, si es necesario resolver problemas como una importación circular o al intentar reducir el tiempo de inicialización de un módulo. Esta técnica es especialmente útil si muchas de las importaciones no son necesarias dependiendo de cómo se ejecute el programa. También podrías mover importaciones a una función si los módulos solo se usan dentro de esa función. Nótese que la primera carga de un módulo puede ser costosa debido al tiempo necesario para la inicialización del módulo,pero la carga de un módulo múltiples veces está prácticamente libre de coste ya que solo es necesario hacer búsquedas en un diccionario. Incluso si el nombre del módulo ha salido del ámbito de aplicación el módulo se encuentre, probablemente, en `sys.modules`.

2.2.6 ¿Por qué los valores por defecto se comparten entre objetos?

Este tipo de error golpea a menudo a programadores novatos. Considera esta función:

```
def foo(mydict={}):  # Danger: shared reference to one dict for all calls
    ... compute something ...
    mydict[key] = value
    return mydict
```

La primera vez que llamas a esta función, `mydict` solamente contiene un único elemento. La segunda vez, `mydict` contiene dos elementos debido a que cuando comienza la ejecución de `foo()`, `mydict` comienza conteniendo un elemento de partida.

A menudo se esperaría que una invocación a una función cree nuevos objetos para valores por defecto. Eso no es lo que realmente sucede. Los valores por defecto se crean exactamente una sola vez, cuando se define la función. Se se cambia el objeto, como el diccionario en este ejemplo, posteriores invocaciones a la función estarán referidas al objeto cambiado.

Por definición, los objetos inmutables como números, cadenas, tuplas y `None` están asegurados frente al cambio. Cambios en objetos mutables como diccionarios, listas e instancias de clase pueden llevar a confusión.

Debido a esta característica es una buena práctica de programación el no usar valores mutables como valores por defecto. En su lugar usa `None` como valor por defecto dentro de la función, comprueba si el parámetro es `None` y crea una nueva lista/un nuevo diccionario/cualquier otras cosa que necesites. Por ejemplo, no escribas:

```
def foo(mydict={}):
    ...
```

pero:

```
def foo(mydict=None):
    if mydict is None:
        mydict = {}  # create a new dict for local namespace
```

Esta característica puede ser útil. Cuando tienes una función que es muy costosa de ejecutar, una técnica común es *cachear* sus parámetros y el valor resultante de cada invocación a la función y retornar el valor *cacheado* si se solicita nuevamente el mismo valor. A esto se le llama «memoizing» y se puede implementar de la siguiente forma:

```
# Callers can only provide two parameters and optionally pass _cache by keyword
def expensive(arg1, arg2, *, _cache={}):
    if (arg1, arg2) in _cache:
        return _cache[(arg1, arg2)]

    # Calculate the value
    result = ... expensive computation ...
    _cache[(arg1, arg2)] = result           # Store result in the cache
    return result
```

Podrías usar una variable global conteniendo un diccionario en lugar de un valor por defecto; es una cuestión de gustos.

2.2.7 ¿Cómo puedo pasar parámetros por palabra clave u opcionales de una función a otra?

Recopila los argumentos usando los especificadores `*` y `**` en la lista de parámetros de la función; esto te proporciona los argumentos posicionales como una tupla y los argumentos con palabras clave como un diccionario. Puedes, entonces, pasar estos argumentos cuando invoques a otra función usando `*` y `**`:

```
def f(x, *args, **kwargs):
    ...
    kwargs['width'] = '14.3c'
    ...
    g(x, *args, **kwargs)
```

2.2.8 ¿Cuál es la diferencia entre argumentos y parámetros?

Parámetros se definen mediante los nombres que aparecen en la definición de una función mientras que *argumentos* son los valores que se pasan a la función cuando la invocamos. Los Parámetros definen qué tipos de argumentos puede aceptar una función. por ejemplo, dada la definición de la función:

```
def func(foo, bar=None, **kwargs):
    pass
```

foo, *bar* y *kwargs* son parámetros de `func`. Sin embargo, cuando invocamos a `func`, por ejemplo:

```
func(42, bar=314, extra=somevar)
```

los valores `42`, `314` y `somevar` son argumentos.

2.2.9 ¿Por qué cambiando la lista "y" cambia, también, la lista "x"?

Si escribes código como:

```
>>> x = []
>>> y = x
>>> y.append(10)
>>> y
[10]
>>> x
[10]
```

te estarás preguntando porque añadir un elemento a `y` ha cambiado también a `x`.

Hay dos factores que provocan este resultado:

1) Las variables son simplemente nombres que referencian a objetos. Haciendo `y = x` no crea una copia de la lista – crea una nueva variable `y` que referencia al mismo objeto al que referencia `x` . Esto significa que solo existe un objeto (la lista) y tanto `x` como `y` hacen referencia al mismo.

2) Las listas son *mutable*, lo que significa que puedes cambiar su contenido.

Después de la invocación a `append()`, el contenido del objeto mutable ha cambiado de `[]` a"[10]". Ya que ambas variables referencian al mismo objeto, el usar cualquiera de los nombres accederá al valor modificado `[10]`.

Si, por otra parte, asignamos un objeto inmutable a `x`:

```
>>> x = 5  # ints are immutable
>>> y = x
>>> x = x + 1  # 5 can't be mutated, we are creating a new object here
>>> x
6
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> y
5
```

podemos ver que `x` e `y` ya no son iguales. Esto es debido a que los enteros son *immutable*, y cuando hacemos `x = x + 1` no estamos mutando el entero `5` incrementando su valor; en su lugar, estamos creando un nuevo objeto (el entero `6`) y se lo asignamos a `x` (esto es, cambiando el objeto al cual referencia `x`). Después de esta asignación tenemos dos objetos (los enteros `6` y `5`) y dos variables que referencian a ellos (`x` ahora referencia a `6` pero `y` todavía referencia a `5`).

Algunas operaciones (por ejemplo `y.append(10)` y `y.sort()`) mutan al objeto mientras que operaciones que podrían parecer similares (por ejemplo `y = y + [10]` y `sorted(y)`) crean un nuevo objeto. En general, en Python (y en todo momento en la biblioteca estándar) un método que muta un objeto retornará `None` para evitar tener dos tipos de operaciones que puedan ser confusas. Por tanto, si escribes accidentalmente `y.sort()` pensando que te retornará una copia ordenada de `y`, obtendrás, en su lugar, `None`, lo cual ayudará a que tu programa genera un error que pueda ser diagnosticado fácilmente.

Sin embargo, existe una clase de operaciones en las cuales la misma operación tiene, a veces, distintos comportamientos con diferentes tipos: los operadores de asignación aumentada. Por ejemplo, += muta listas pero no tuplas o enteros (`a_list += [1, 2, 3]` es equivalente a `a_list.extend([1, 2, 3])` y muta `a_list`, mientras que `some_tuple += (1, 2, 3)` y `some_int += 1` crea nuevos objetos).

En otras palabras:

- Si tenemos un objeto mutable (`list`, `dict`, `set`, etc.), podemos usar algunas operaciones específicas para mutarlo y todas las variables que referencian al mismo verán el cambio reflejado.
- Si tenemos un objeto inmutable (`str`, `int`, `tuple`, etc.), todas las variables que referencian al mismo verán siempre el mismo valor pero las operaciones que transforman ese valor en un nuevo valor siempre retornan un nuevo objeto.

Si deseas saber si dos variables referencian o no al mismo objeto puedes usar el operador `is` o la función incorporada `id()`.

2.2.10 ¿Cómo puedo escribir una función sin parámetros (invocación mediante referencia)?

Recuerda que los argumentos son pasados mediante asignación en Python. Ya que las asignaciones simplemente crean referencias a objetos, no hay alias entre el nombre de un argumento en el invocador y el invocado y, por tanto, no hay invocación por referencia per se. Puedes obtener el mismo efecto deseado de formas distintas.

1) Mediante el retorno de una tupla de resultados:

```
>>> def func1(a, b):
...     a = 'new-value'        # a and b are local names
...     b = b + 1              # assigned to new objects
...     return a, b            # return new values
...
>>> x, y = 'old-value', 99
>>> func1(x, y)
('new-value', 100)
```

   Esta es, casi siempre, la solución más clara.

2) Mediante el uso de variables globales. No es thread-safe y no se recomienda.

3) Pasando un objeto mutable (intercambiable en el mismo sitio):

```
>>> def func2(a):
...     a[0] = 'new-value'     # 'a' references a mutable list
...     a[1] = a[1] + 1        # changes a shared object
...
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> args = ['old-value', 99]
>>> func2(args)
>>> args
['new-value', 100]
```

4) Pasando un diccionario que muta:

```
>>> def func3(args):
...     args['a'] = 'new-value'     # args is a mutable dictionary
...     args['b'] = args['b'] + 1   # change it in-place
...
>>> args = {'a': 'old-value', 'b': 99}
>>> func3(args)
>>> args
{'a': 'new-value', 'b': 100}
```

5) O empaquetar valores en una instancia de clase:

```
>>> class Namespace:
...     def __init__(self, /, **args):
...         for key, value in args.items():
...             setattr(self, key, value)
...
>>> def func4(args):
...     args.a = 'new-value'        # args is a mutable Namespace
...     args.b = args.b + 1         # change object in-place
...
>>> args = Namespace(a='old-value', b=99)
>>> func4(args)
>>> vars(args)
{'a': 'new-value', 'b': 100}
```

Casi nunca existe una buena razón para hacer esto tan complicado.

Tu mejor opción es retornar una tupla que contenga los múltiples resultados.

2.2.11 ¿Cómo se puede hacer una función de orden superior en Python?

Tienes dos opciones: puedes usar ámbitos de aplicación anidados o puedes usar objetos invocables. Por ejemplo, supón que querías definir `linear(a,b)` que devuelve una función `f(x)` que calcula el valor `a*x+b`. Usar ámbitos de aplicación anidados:

```
def linear(a, b):
    def result(x):
        return a * x + b
    return result
```

O usar un objeto invocable:

```
class linear:

    def __init__(self, a, b):
        self.a, self.b = a, b

    def __call__(self, x):
        return self.a * x + self.b
```

En ambos casos,

```
taxes = linear(0.3, 2)
```

nos da un objeto invocable donde `taxes(10e6) == 0.3 * 10e6 + 2`.

El enfoque del objeto invocable tiene la desventaja que es un ligeramente más lento y el resultado es un código levemente más largo. Sin embargo, destacar que una colección de invocables pueden compartir su firma vía herencia:

```
class exponential(linear):
    # __init__ inherited
    def __call__(self, x):
        return self.a * (x ** self.b)
```

Los objetos pueden encapsular el estado de varios métodos:

```
class counter:

    value = 0

    def set(self, x):
        self.value = x

    def up(self):
        self.value = self.value + 1

    def down(self):
        self.value = self.value - 1

count = counter()
inc, dec, reset = count.up, count.down, count.set
```

Aquí `inc()`, `dec()` y `reset()` se comportan como funciones las cuales comparten la misma variable de conteo.

2.2.12 ¿Cómo copio un objeto en Python?

En general, prueba `copy.copy()` o `copy.deepcopy()` para el caso general. No todos los objetos se pueden copiar pero la mayoría sí que pueden copiarse.

Algunas objetos se pueden copiar de forma más sencilla. Los diccionarios disponen de un método `copy()`:

```
newdict = olddict.copy()
```

Las secuencias se pueden copiar usando un rebanado:

```
new_l = l[:]
```

2.2.13 ¿Cómo puedo encontrar los métodos o atributos de un objeto?

Para la instancia x de una clase definida por el usuario, `dir(x)` devuelve una lista de nombres ordenados alfabéticamente que contiene los atributos y métodos de la instancia y los atributos definidos mediante su clase.

2.2.14 ¿Cómo puede mi código descubrir el nombre de un objeto?

Hablando de forma general no podrían puesto que los objetos no disponen, realmente, de un nombre. Esencialmente, las asignaciones relacionan un nombre con su valor; Lo mismo se cumple con las declaraciones `def` y `class` pero, en este caso, el valor es un invocable. Considera el siguiente código:

```
>>> class A:
...     pass
...
>>> B = A
>>> a = B()
>>> b = a
>>> print(b)
<__main__.A object at 0x16D07CC>
>>> print(a)
<__main__.A object at 0x16D07CC>
```

Podría decirse que la clase tiene un nombre: aunque está ligada a dos nombres y se invoca a través del nombre B, la instancia creada se sigue reportando como una instancia de la clase A. Sin embargo, es imposible decir si el nombre de la instancia es a o b, ya que ambos nombres están ligados al mismo valor.

En términos generales, no debería ser necesario que tu código «conozca los nombres» de determinados valores. A menos que estés escribiendo deliberadamente programas introspectivos, esto suele ser una indicación de que un cambio de enfoque podría ser beneficioso.

En comp.lang.python, Fredrik Lundh proporcionó una vez una excelente analogía en respuesta a esta pregunta:

> De la misma forma que obtienes el nombre de ese gato que te has encontrado en tu porche el propio gato (objeto) no te puede indicar su nombre y, realmente, no importa – por tanto, la única forma de encontrar cómo se llama sería preguntando a todos los vecinos (espacios de nombres) si es su gato (objeto)…
>
> …y no te sorprendas si encuentras que se le conoce mediante diferentes nombres o ¡nadie conoce su nombre!

2.2.15 ¿Qué ocurre con la precedencia del operador coma?

La coma no es un operador en Python. Considera la sesión:

```
>>> "a" in "b", "a"
(False, 'a')
```

Debido a que la coma no es un operador sino un separador entre expresiones lo anterior se evalúe como se ha introducido:

```
("a" in "b"), "a"
```

no:

```
"a" in ("b", "a")
```

Lo mismo sucede con varios operadores de asignación (=, +=, etc). No son realmente operadores sino delimitadores sintácticos en declaraciones de asignación.

2.2.16 ¿Existe un equivalente al operador ternario de C «?:»?

Sí, existe. La sintaxis es como sigue:

```
[on_true] if [expression] else [on_false]

x, y = 50, 25
small = x if x < y else y
```

Antes de que esta sintaxis se introdujera en Python 2.5 una expresión común fue el uso de operadores lógicos:

```
[expression] and [on_true] or [on_false]
```

Sin embargo, esa expresión no es segura ya que puede retornar valores erróneos cuando *on_true* tiene un valor booleano falso. Por tanto, siempre es mejor usar la forma `... if ... else ...`.

2.2.17 ¿Es posible escribir expresiones en una línea de forma ofuscada en Python?

Sí. Normalmente se puede hacer anidando `lambda` dentro de `lambda`. Examina los siguientes tres ejemplos, creados por Ulf Bartelt:

```
from functools import reduce

# Primes < 1000
print(list(filter(None,map(lambda y:y*reduce(lambda x,y:x*y!=0,
map(lambda x,y=y:y%x,range(2,int(pow(y,0.5)+1))),1),range(2,1000)))))

# First 10 Fibonacci numbers
print(list(map(lambda x,f=lambda x,f:(f(x-1,f)+f(x-2,f)) if x>1 else 1:
f(x,f), range(10))))

# Mandelbrot set
print((lambda Ru,Ro,Iu,Io,IM,Sx,Sy:reduce(lambda x,y:x+y,map(lambda y,
Iu=Iu,Io=Io,Ru=Ru,Ro=Ro,Sy=Sy,L=lambda yc,Iu=Iu,Io=Io,Ru=Ru,Ro=Ro,i=IM,
Sx=Sx,Sy=Sy:reduce(lambda x,y:x+y,map(lambda x,xc=Ru,yc=yc,Ru=Ru,Ro=Ro,
i=i,Sx=Sx,F=lambda xc,yc,x,y,k,f=lambda xc,yc,x,y,k,f:(k<=0)or (x*x+y*y
>=4.0) or 1+f(xc,yc,x*x-y*y+xc,2.0*x*y+yc,k-1,f):f(xc,yc,x,y,k,f):chr(
64+F(Ru+x*(Ro-Ru)/Sx,yc,0,0,i)),range(Sx))):L(Iu+y*(Io-Iu)/Sy),range(Sy
))))(-2.1, 0.7, -1.2, 1.2, 30, 80, 24))
#     \___ ___/  \___ ___/  |   |   |__ lines on screen
#         V          V      |   |______ columns on screen
#         |          |      |_________ maximum of "iterations"
#         |          |________________ range on y axis
#         |___________________________ range on x axis
```

¡No probéis esto en casa, personitas!

2.2.18 ¿Qué hace la barra (/) en medio de la lista de parámetros de una función?

Un *slash* en la lista de argumentos de una función denota que los parámetros previos al mismo son únicamente posicionales. Parámetros únicamente posicionales son aquellos cuyos nombres no son usables internamente. Mediante la llamada a una función que acepta parámetros únicamente posicionales, los argumentos se mapean a parámetros basados únicamente en su posición. Por ejemplo, `pow()` es una función que acepta parámetros únicamente posicionales. Su documentación es de la siguiente forma:

```
>>> help(divmod)
Help on built-in function divmod in module builtins:

```

(continué en la próxima página)

(proviene de la página anterior)

```
divmod(x, y, /)
    Return the tuple (x//y, x%y).  Invariant: div*y + mod == x.
```

El *slash* al final de la lista de parámetros indica que los tres parámetros son únicamente posicionales. Por tanto, invocar a `pow()` con argumentos con palabra clave podría derivar en un error:

```
>>> divmod(x=3, y=4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: divmod() takes no keyword arguments
```

2.3 Números y cadenas

2.3.1 ¿Cómo puedo especificar enteros hexadecimales y octales?

Para especificar un dígito octal, prefija el valor octal con un cero y una «o» en minúscula o mayúscula. Por ejemplo, para definir la variable «a» con el valor octal «10» (8 en decimal), escribe:

```
>>> a = 0o10
>>> a
8
```

Un hexadecimal es igual de simple. Simplemente añade un cero y una «x», en minúscula o mayúscula, antes del número hexadecimal . Los dígitos hexadecimales se pueden especificar en minúsculas o mayúsculas. Por ejemplo, en el intérprete de Python:

```
>>> a = 0xa5
>>> a
165
>>> b = 0XB2
>>> b
178
```

2.3.2 ¿Por qué -22 // 10 devuelve -3?

Es debido, principalmente al deseo que `i % j` tenga el mismo signo que `j`. Si quieres eso y, además, quieres:

```
i == (i // j) * j + (i % j)
```

entonces la división entera debe retornar el valor base más bajo. C también requiere que esa esa identidad se mantenga de tal forma que cuando los compiladores truncan `i // j` necesitan que `i % j` tenga el mismo signo que `i`.

Existen unos pocos casos para `i % j` cuando `j` es negativo. Cuando `j` es positivo, existen muchos casos y, virtualmente, en todos ellos es más útil para `i % j` que sea `>= 0`. Si el reloj dice que ahora son las 10, ¿qué dijo hace 200 horas? `-190 % 12 == 2` es útil; `-190 % 12 == -10` es un error listo para morderte.

2.3.3 ¿Cómo convierto una cadena a un número?

Para enteros puedes usar la función incorporada constructor de tipos `int()`, por ejemplo `int('144') == 144`. De forma similar, `float()` convierte a un número de coma flotante, por ejemplo `float('144') == 144.0`.

Por defecto, estas interpretan el número como decimal de tal forma que `int('0144') == 144` y `int('0x144')` lanzará `ValueError`. `int(string, base)` toma la base para convertirlo desde un segundo parámetro opcional, por tanto `int('0x144', 16) == 324`. Si la base se especifica como 0, el número se interpreta usando las reglas de Python's rules: un prefijo "0o" indica octal y un prefijo "0x" indica un número hexadecimal.

No uses la función incorporada `eval()` si todo lo que necesitas es convertir cadenas a números. `eval()` será considerablemente más lento y presenta riesgos de seguridad: cualquiera podría introducir una expresión Python que presentara efectos indeseados. Por ejemplo, alguien podría pasar"__import__("os").system(«rm -rf $HOME»)" lo cual borraría el directorio home al completo.

`eval()` también tiene el efecto de interpretar números como expresiones Python , de tal forma que, por ejemplo, `eval('09')` dará un error de sintaxis porque Python no permite un "0" inicial en un número decimal (excepto "0").

2.3.4 ¿Cómo puedo convertir un número a una cadena?

Para convertir, por ejemplo, el número 144 a la cadena "144", usa el constructor de tipos incorporado `str()`. Si deseas una representación hexadecimal o octal usa la función incorporada `hex()` o `oct()`. Para un formateado elaborado puedes ver las secciones de f-strings y formatstrings, por ejemplo `"{:04d}".format(144)` produce `'0144'` y `"{:.3f}".format(1.0/3.0)` produce `'0.333'`.

2.3.5 ¿Cómo puedo modificar una cadena in situ?

No puedes debido a que las cadenas son inmutables. En la mayoría de situaciones solo deberías crear una nueva cadena a partir de varias partes que quieras usar para crearla. Sin embargo, si necesitas un objeto con la habilidad de modificar en el mismo lugar datos unicode prueba usando el objeto `io.StringIO` o el módulo `array`:

```
>>> import io
>>> s = "Hello, world"
>>> sio = io.StringIO(s)
>>> sio.getvalue()
'Hello, world'
>>> sio.seek(7)
7
>>> sio.write("there!")
6
>>> sio.getvalue()
'Hello, there!'

>>> import array
>>> a = array.array('u', s)
>>> print(a)
array('u', 'Hello, world')
>>> a[0] = 'y'
>>> print(a)
array('u', 'yello, world')
>>> a.tounicode()
'yello, world'
```

2.3.6 ¿Cómo puedo usar cadenas para invocar funciones/métodos?

Existen varias técnicas.

- Lo mejor sería usar un diccionario que mapee cadenas a funciones. La principal ventaja de esta técnica es que las cadenas no necesitan ser iguales que los nombres de las funciones. Esta es también la principal técnica que se usa para emular un constructo *case*:

```
def a():
    pass

def b():
    pass

dispatch = {'go': a, 'stop': b}  # Note lack of parens for funcs

dispatch[get_input()]()  # Note trailing parens to call function
```

- Usa la función incorporada `getattr()`:

```
import foo
getattr(foo, 'bar')()
```

  Nótese que `getattr()` funciona en cualquier objeto, incluido clases, instancias de clases, módulos, etc.

  Esto se usa en varios lugares de la biblioteca estándar, como esto:

```
class Foo:
    def do_foo(self):
        ...

    def do_bar(self):
        ...

f = getattr(foo_instance, 'do_' + opname)
f()
```

- Usa `locals()` o `eval()` para resolver el nombre de la función:

```
def myFunc():
    print("hello")

fname = "myFunc"

f = locals()[fname]
f()

f = eval(fname)
f()
```

  Nota: Usar `eval()` es lento y peligroso. Si no tienes el control absoluto del contenido de la cadena cualquiera podría introducir una cadena que resulte en la ejecución de código arbitrario.

2.3.7 ¿Existe un equivalente a chomp() en Perl para eliminar nuevas líneas al final de las cadenas?

Puedes usar `S.rstrip("\r\n")` para eliminar todas las ocurrencias de cualquier terminación de línea desde el final de la cadena `S` sin eliminar el resto de espacios en blanco que le siguen. Si la cadena `S` representa más de una línea con varias líneas vacías al final, las terminaciones de línea para todas las líneas vacías se eliminarán:

```
>>> lines = ("line 1 \r\n"
...          "\r\n"
...          "\r\n")
>>> lines.rstrip("\n\r")
'line 1 '
```

Ya que esto solo sería deseable, típicamente, cuando lees texto línea a línea, usar `S.rstrip()` de esta forma funcionaría bien.

2.3.8 ¿Existe un equivalente a scanf() o a sscanf() ?

No de la misma forma.

Para análisis sintáctico simple de la entrada, el método más sencillo es, usualmente, el separar la línea en palabras delimitadas por espacios usando el método `split()` de los objetos *string* y, posteriormente, convertir cadenas decimales a valores usando `int()` o `float()`. `split()` permite un parámetro opcional «sep» que es útil si la línea usa algo diferente a espacios en blanco como separador.

Para análisis sintáctico de la entrada más complejo, las expresiones regulares son más poderosas que `sscanf()` de C y se ajustan mejor a esta tarea.

2.3.9 ¿Qué significa “UnicodeDecodeError” o “UnicodeEncodeError”?

Ver unicode-howto.

2.4 Rendimiento

2.4.1 Mi programa es muy lento. ¿Cómo puedo acelerarlo?

Esa es una pregunta difícil, en general. Primero, aquí tienes una lista de cosas a recordar antes de ir más allá:

- Las características del rendimiento varían entre las distintas implementaciones de Python. Estas preguntas frecuentes se enfocan en *CPython*.
- El comportamiento puede variar entre distintos sistemas operativos, especialmente cuando se habla de tareas I/O o multi-tarea.
- Siempre deberías encontrar las partes importantes en tu programa *antes* de intentar optimizar el código (ver el módulo `profile`).
- Escribir programas de comparación del rendimiento te permitirá iterar rápidamente cuando te encuentres buscando mejoras (ver el módulo `timeit`).
- Es altamente recomendable disponer de una buena cobertura de código (a partir de pruebas unitarias o cualquier otra técnica) antes de introducir potenciales regresiones ocultas en sofisticadas optimizaciones.

Dicho lo anterior, existen muchos trucos para acelerar código Python. Aquí tienes algunos principios generales que te permitirán llegar a alcanzar niveles de rendimiento aceptables:

- El hacer más rápido tu algoritmo (o cambiarlo por alguno más rápido) puede provocar mayores beneficios que intentar unos pocos trucos de micro-optimización a través de todo tu código.

- Utiliza las estructuras de datos correctas. Estudia la documentación para los bltin-types y el módulo `collections`.
- Cuando la biblioteca estándar proporciona una primitiva de hacer algo, esta supuestamente será (aunque no se garantiza) más rápida que cualquier otra alternativa que se te ocurra. Esto es doblemente cierto si las primitivas han sido escritas en C, como los *builtins* y algunos tipos extendidos. Por ejemplo, asegúrate de usar el método integrado `list.sort()` o la función relacionada `sorted()` para ordenar (y ver sortinghowto para ver ejemplos de uso moderadamente avanzados).
- Las abstracciones tienden a crear rodeos y fuerzan al intérprete a trabajar más. Si el nivel de rodeos sobrepasa el trabajo útil realizado tu programa podría ser más lento. Deberías evitar abstracciones excesivas, especialmente, en forma de pequeñas funciones o métodos (que también va en detrimento de la legibilidad).

Si has alcanzado el límite de lo que permite el uso de Python puro, existen otras herramientas que te permiten ir más allá. Por ejemplo, Cython puede compilar una versión ligeramente modificada del código Python en una extensión en C y se podría usar en muchas plataformas diferentes. Cython puede obtener ventaja de la compilación (y anotaciones de tipos opcionales) para hacer que tu código sea significativamente más rápido cuando se ejecuta. Si confías en tus habilidades de programar en C también puedes escribir un módulo de extensión en C tú mismo.

Ver también:

La página de la wiki dedicada a trucos de rendimiento.

2.4.2 ¿Cuál es la forma más eficiente de concatenar muchas cadenas conjuntamente?

Los objetos `str` y `bytes` son inmutables, por tanto, concatenar muchas cadenas en una sola es ineficiente debido a que cada concatenación crea un nuevo objeto. En el caso más general, el coste total en tiempo de ejecución es cuadrático en relación a la longitud de la cadena final.

Para acumular muchos objetos `str`, la forma recomendada sería colocarlos en una lista y llamar al método `str.join()` al final:

```
chunks = []
for s in my_strings:
    chunks.append(s)
result = ''.join(chunks)
```

(otra forma que sería razonable en términos de eficiencia sería usar `io.StringIO`)

Para acumular muchos objetos `bytes`, la forma recomendada sería extender un objeto `bytearray` usando el operador de concatenación in situ (el operador +=):

```
result = bytearray()
for b in my_bytes_objects:
    result += b
```

2.5 Secuencias (Tuplas/Listas)

2.5.1 ¿Cómo convertir entre tuplas y listas?

El constructor `tuple(seq)` convierte cualquier secuencia (en realidad, cualquier iterable) en una tupla con los mismos elementos y en el mismo orden.

Por ejemplo, `tuple([1, 2, 3])` lo convierte en `(1, 2, 3)` y `tuple('abc')` lo convierte en `('a', 'b', 'c')`. Si el argumento es una tupla no creará una nueva copia y retornará el mismo objeto, por tanto, llamar a `tuple()` no tendrá mucho coste si no estás seguro si un objeto ya es una tupla.

El constructor `list(seq)` convierte cualquier secuencia o iterable en una lista con los mismos elementos y en el mismo orden. Por ejemplo, `list((1, 2, 3))` lo convierte a `[1, 2, 3]` y `list('abc')` lo convierte a `['a', 'b', 'c']`. Si el argumento es una lista, hará una copia como lo haría `seq[:]`.

2.5.2 ¿Qué es un índice negativo?

Las secuencias en Python están indexadas con números positivos y negativos. Para los números positivos el 0 será el primer índice, el 1 el segundo y así en adelante. Para los índices negativos el -1 el último índice, el -2 el penúltimo, etc. Piensa en `seq[-n]` como si fuera `seq[len(seq)-n]`.

El uso de índices negativos puede ser muy conveniente. Por ejemplo `S[:-1]` se usa para todo la cadena excepto para su último carácter, lo cual es útil para eliminar el salto de línea final de una cadena.

2.5.3 ¿Cómo puedo iterar sobre una secuencia en orden inverso?

Use the `reversed()` built-in function:

```
for x in reversed(sequence):
    ...  # do something with x ...
```

Esto no transformará la secuencia original sino que creará una nueva copia en orden inverso por la que se puede iterar.

2.5.4 ¿Cómo eliminar duplicados de una lista?

Puedes echar un vistazo al recetario de Python para ver una gran discusión mostrando muchas formas de hacer esto:

https://code.activestate.com/recipes/52560/

Si no te preocupa que la lista se reordene la puedes ordenar y, después, y después escanearla desde el final borrando duplicados a medida que avanzas:

```
if mylist:
    mylist.sort()
    last = mylist[-1]
    for i in range(len(mylist)-2, -1, -1):
        if last == mylist[i]:
            del mylist[i]
        else:
            last = mylist[i]
```

Si todos los elementos de la lista pueden ser usados como claves (por ejemplo son todos *hashable*) esto será, en general, más rápido

```
mylist = list(set(mylist))
```

Esto convierte la lista en un conjunto eliminando, por tanto, los duplicados y, posteriormente, puedes volver a una lista.

2.5.5 How do you remove multiple items from a list

As with removing duplicates, explicitly iterating in reverse with a delete condition is one possibility. However, it is easier and faster to use slice replacement with an implicit or explicit forward iteration. Here are three variations.:

```
mylist[:] = filter(keep_function, mylist)
mylist[:] = (x for x in mylist if keep_condition)
mylist[:] = [x for x in mylist if keep_condition]
```

The list comprehension may be fastest.

2.5.6 ¿Cómo se puede hacer un array en Python?

Usa una lista:

```
["this", 1, "is", "an", "array"]
```

Las listas son equivalentes en complejidad temporal a arrays en C o Pascal; La principal diferencia es que una lista en Python puede contener objetos de diferentes tipos.

El módulo `array` proporciona, también, métodos para crear arrays de tipo fijo con representaciones compactas pero son más lentos de indexar que las listas. Además, debes tener en cuenta que las extensiones Numeric y otras permiten definen estructuras de tipo array con diversas características.

Para obtener listas enlazadas al estilo de las de Lisp, puedes emular celdas cons usando tuplas:

```
lisp_list = ("like",  ("this",  ("example", None) ) )
```

Si deseas que haya mutabilidad podrías usar listas en lugar de tuplas. El análogo a un car de Lisp es `lisp_list[0]` y al análogo a cdr es `lisp_list[1]`. Haz esto solo si estás seguro que es lo que necesitas debido a que, normalmente, será bastante más lento que el usar listas Python.

2.5.7 ¿Cómo puedo crear una lista multidimensional?

Seguramente hayas intentado crear un array multidimensional de la siguiente forma:

```
>>> A = [[None] * 2] * 3
```

Esto parece correcto si lo muestras en pantalla:

```
>>> A
[[None, None], [None, None], [None, None]]
```

Pero cuando asignas un valor, se muestra en múltiples sitios:

```
>>> A[0][0] = 5
>>> A
[[5, None], [5, None], [5, None]]
```

La razón es que replicar una lista con `*` no crea copias, solo crea referencias a los objetos existentes. El `*3` crea una lista conteniendo 3 referencias a la misma lista de longitud dos. Cambios a una fila se mostrarán en todas las filas, lo cual, seguramente, no es lo que deseas.

El enfoque recomendado sería crear, primero, una lista de la longitud deseada y, después, rellenar cada elemento con una lista creada en ese momento:

```
A = [None] * 3
for i in range(3):
    A[i] = [None] * 2
```

Esto genera una lista conteniendo 3 listas distintas de longitud dos. También puedes usar una comprensión de lista:

```
w, h = 2, 3
A = [[None] * w for i in range(h)]
```

O puedes usar una extensión que proporcione un tipo de dato para matrices; NumPy es la más conocida.

2.5.8 ¿Cómo puedo aplicar un método a una secuencia de objetos?

Usa una comprensión de listas:

```
result = [obj.method() for obj in mylist]
```

2.5.9 ¿Por qué hacer lo siguiente, `a_tuple[i] += ['item']`, lanza una excepción cuando la suma funciona?

Esto es debido a la combinación del hecho de que un operador de asignación aumentada es un operador de *asignación* y a la diferencia entre objetos mutables e inmutable en Python.

Esta discusión aplica, en general, cuando los operadores de asignación aumentada se aplican a elementos de una tupla que apuntan a objetos mutables. Pero vamos a usar una `lista` y += para el ejemplo.

Si escribes:

```
>>> a_tuple = (1, 2)
>>> a_tuple[0] += 1
Traceback (most recent call last):
  ...
TypeError: 'tuple' object does not support item assignment
```

La razón por la que se produce la excepción debería ser evidente: `1` se añade al objeto `a_tuple[0]` que apunta a (`1`), creando el objeto resultante, `2`, pero cuando intentamos asignar el resultado del cálculo, `2`, al elemento `0` de la tupla, obtenemos un error debido a que no podemos cambiar el elemento al que apunta la tupla.

En realidad, lo que esta declaración de asignación aumentada está haciendo es, aproximadamente, lo siguiente:

```
>>> result = a_tuple[0] + 1
>>> a_tuple[0] = result
Traceback (most recent call last):
  ...
TypeError: 'tuple' object does not support item assignment
```

Es la parte de asignación de la operación la que provoca el error, debido a que una tupla es inmutable.

Cuando escribes algo como lo siguiente:

```
>>> a_tuple = (['foo'], 'bar')
>>> a_tuple[0] += ['item']
Traceback (most recent call last):
  ...
TypeError: 'tuple' object does not support item assignment
```

La excepción es un poco más sorprendente e, incluso, más sorprendente es el hecho que aunque hubo un error, la agregación funcionó:

```
>>> a_tuple[0]
['foo', 'item']
```

Para ver lo que sucede necesitas saber que (a) si un objeto implementa un método mágico `__iadd__` , se le llama cuando se ejecuta la asignación aumentada += y el valor devuelto es lo que se usa en la declaración de asignación; y

(b) para listas, `__iadd__` es equivalente a llamar a `extend` en la lista y retornar la lista. Es por esto que decimos que para listas, += es un atajo para `list.extend`:

```
>>> a_list = []
>>> a_list += [1]
>>> a_list
[1]
```

Esto es equivalente a

```
>>> result = a_list.__iadd__([1])
>>> a_list = result
```

El objeto al que apunta a_list ha mutado y el puntero al objeto mutado es asignado de vuelta a `a_list`. El resultado final de la asignación no es opción debido a que es un puntero al mismo objeto al que estaba apuntando `a_list` pero la asignación sí que ocurre.

Por tanto, en nuestro ejemplo con tupla lo que está pasando es equivalente a:

```
>>> result = a_tuple[0].__iadd__(['item'])
>>> a_tuple[0] = result
Traceback (most recent call last):
  ...
TypeError: 'tuple' object does not support item assignment
```

El `__iadd__` se realiza con éxito y la lista se extiende pero, incluso aunque `result` apunta al mismo objeto al que ya está apuntando `a_tuple[0]` la asignación final sigue resultando en un error, debido a que las tuplas son inmutables.

2.5.10 Quiero hacer una ordenación compleja: ¿Puedes hacer una transformada Schwartziana (Schwartzian Transform) en Python?

La técnica, atribuida a Randal Schwartz, miembro de la comunidad Perl, ordena los elementos de una lista mediante una métrica que mapea cada elemento a su «valor orden». En Python, usa el argumento `key` par el método `list.sort()`:

```
Isorted = L[:]
Isorted.sort(key=lambda s: int(s[10:15]))
```

2.5.11 ¿Cómo puedo ordenar una lista a partir de valores de otra lista?

Las puedes unir en un iterador de tuplas, ordena la lista resultando y después extrae el elemento que deseas.

```
>>> list1 = ["what", "I'm", "sorting", "by"]
>>> list2 = ["something", "else", "to", "sort"]
>>> pairs = zip(list1, list2)
>>> pairs = sorted(pairs)
>>> pairs
[("I'm", 'else'), ('by', 'sort'), ('sorting', 'to'), ('what', 'something')]
>>> result = [x[1] for x in pairs]
>>> result
['else', 'sort', 'to', 'something']
```

Una alternativa para el último paso es:

```
>>> result = []
>>> for p in pairs: result.append(p[1])
```

Si encuentras esto más legible, podrías preferir usar esto en lugar de una comprensión de lista final. Sin embargo, es casi el doble de lento para listas largas. ¿Por qué? Primero, la operación `append()` necesita reasignar memoria y, aunque usa algunos trucos para evitar hacerlo en todo momento, sigue teniéndolo que hacer ocasionalmente y eso tiene un poco de coste. Segundo, la expresión «result.append» requiere una búsqueda adicional de atributos y, tercero, hay una reducción de velocidad de tener que hacer todas esas llamadas a funciones.

2.6 Objetos

2.6.1 ¿Qué es una clase?

Una clase es un tipo de objeto particular creado mediante la ejecución de la declaración class. Los objetos class se usan como plantillas para crear instancias de objetos que son tanto los datos (atributos) como el código (métodos) específicos para un tipo de dato.

Una clase puede estar basada en una o más clases diferentes, llamadas su(s) clase(s). Hereda los atributos y métodos de sus clases base. Esto permite que se pueda refinar un objeto modelo de forma sucesiva mediante herencia. Puedes tener una clase genérica `Mailbox` que proporciona métodos de acceso básico para un buzón de correo y subclases como `MboxMailbox`, `MaildirMailbox`, `OutlookMailbox` que gestionan distintos formatos específicos de buzón de correos.

2.6.2 ¿Qué es un método?

Un método es una función de un objeto `x` que puedes llamar, normalmente, de la forma `x.name(arguments...)`. Los métodos se definen como funciones dentro de la definición de la clase:

```
class C:
    def meth(self, arg):
        return arg * 2 + self.attribute
```

2.6.3 ¿Qué es self?

Self es, básicamente, un nombre que se usa de forma convencional como primer argumento de un método. Un método definido como `meth(self, a, b, c)` se le llama como `x.meth(a, b, c)` para una instancia `x` de la clase es que se definió; el método invocado pensará que se le ha invocado como `meth(x, a, b, c)`.

Ver también *¿Por qué debe usarse "self" explícitamente en las definiciones y llamadas de métodos?*.

2.6.4 ¿Cómo puedo comprobar si un objeto es una instancia de una clase dada o de una subclase de la misma?

Usa la función incorporada `isinstance(obj, cls)`. Puedes comprobar si un objeto es una instancia de cualquier número de clases proporcionando una tupla en lugar de una sola clase, por ejemplo `isinstance(obj, (class1, class2, ...))` y, también, puedes comprobar si un objeto es uno de los tipos incorporados por ejemplo `isinstance(obj, str)` o `isinstance(obj, (int, float, complex))`.

Destacar que muchos programas no necesitan usar `isinstance()` de forma frecuente en clases definidas por el usuario. Si estás desarrollando clases un mejor estilo orientado a objetos sería el de definir los métodos en las clases que encapsulan un comportamiento en particular en lugar de ir comprobando la clase del objeto e ir haciendo cosas en base a la clase que es. Por ejemplo, si tienes una función que hace lo siguiente:

```
def search(obj):
    if isinstance(obj, Mailbox):
        ...  # code to search a mailbox
    elif isinstance(obj, Document):
```

(continué en la próxima página)

(proviene de la página anterior)

```
        ...  # code to search a document
    elif ...
```

Un enfoque más adecuado sería definir un método `search()` en todas las clases e invocarlo:

```
class Mailbox:
    def search(self):
        ...  # code to search a mailbox

class Document:
    def search(self):
        ...  # code to search a document

obj.search()
```

2.6.5 ¿Qué es la delegación?

La delegación es una técnica orientada a objetos (también llamado un patrón de diseño). Digamos que tienes un objeto `x` y deseas cambiar el comportamiento de solo uno de sus métodos. Puedes crear una nueva clase que proporciona una nueva implementación del método que te interesa cambiar y delega el resto de métodos al método correspondiente de `x`.

Los programadores Python pueden implementar la delegación de forma muy sencilla. Por ejemplo, la siguiente clase implementa una clase que se comporta como un fichero pero convierte todos los datos escritos a mayúsculas:

```
class UpperOut:

    def __init__(self, outfile):
        self._outfile = outfile

    def write(self, s):
        self._outfile.write(s.upper())

    def __getattr__(self, name):
        return getattr(self._outfile, name)
```

Aquí, la clase `UpperOut` redefine el método `write()` para convertir la cadena del argumento a mayúscula antes de invocar al método `self._outfile.write()`. El resto de métodos han sido delegados al objeto `self._outfile`. La delegación se consigue mediante el método `__getattr__`; consulta la referencia del lenguaje para obtener más información sobre cómo controlar el acceso a atributos.

Ten en cuenta que para casos más generales la delegación puede ser algo más complicada. Cuando los atributos se deben colocar y recuperar la clase debe definir, también, un método `__setattr__()` y hay que hacerlo con cuidado. La implementación básica de `__setattr__()` es, aproximadamente, equivalente a lo siguiente:

```
class X:
    ...
    def __setattr__(self, name, value):
        self.__dict__[name] = value
    ...
```

Muchas implementaciones de `__setattr__()` deben modificar `self.__dict__` para almacenar el estado local para self sin provocar una recursión infinita.

2.6.6 ¿Cómo invoco a un método definido en una clase base desde una clase derivada que lo ha sobreescrito?

Usa la función incorporada `super()`:

```
class Derived(Base):
    def meth(self):
        super(Derived, self).meth()
```

Para versiones anteriores a la 3.0, puedes usar clases clásicas: Para la definición de una clase como `class Derived(Base): ...` puedes invocar el método `meth()` definido en `Base` (o una de las clases base de `Base`) como `Base.meth(self, arguments...)`. Aquí, `Base.meth` es un método no ligado y, por tanto, debes proporcionar el argumento `self`.

2.6.7 ¿Cómo puedo organizar mi código para hacer que sea más sencillo modificar la clase base?

Puedes definir un alias para la clase base, asignar la clase base real al alias antes de la definición de tu clase y usar el alias a lo largo de toda la clase. Entonces, lo único que tienes que cambiar es el valor asignado al alias. Sin pretenderlo, este truco también es útil si desear decidir de forma dinámica (por ejemplo dependiendo de la disponibilidad de recursos) qué clase base usar. Ejemplo:

```
BaseAlias = <real base class>

class Derived(BaseAlias):
    def meth(self):
        BaseAlias.meth(self)
        ...
```

2.6.8 ¿Cómo puedo crear datos estáticos de clase y métodos estáticos de clase?

Tanto los datos estáticos como los métodos estáticos (en el sentido de C++ o Java) están permitidos en Python.

Para datos estáticos simplemente define un atributo de clase. Para asignar un nuevo valor al atributo debes usar de forma explícita el nombre de la clase en la asignación:

```
class C:
    count = 0   # number of times C.__init__ called

    def __init__(self):
        C.count = C.count + 1

    def getcount(self):
        return C.count  # or return self.count
```

`c.count` también se refiere a `C.count` para cualquier `c` de tal forma que se cumpla `isinstance(c, C)`, a no ser que `c` sea sobreescrita por si misma o por alguna clase contenida en la búsqueda de clases base desde“c.__class__“ hasta `C`.

Debes tener cuidado: dentro de un método de C, una asignación como `self.count = 42` creará una nueva instancia sin relación con la original que se llamará «count» en el propio diccionario de `self`. El reunificar el nombre de datos estáticos de una clase debería llevar, siempre, a especificar la clase tanto si se produce desde dentro de un método como si no:

```
C.count = 314
```

Los métodos estáticos son posibles:

```
class C:
    @staticmethod
    def static(arg1, arg2, arg3):
        # No 'self' parameter!
        ...
```

Sin embargo, una forma más directa de obtener el efecto de un método estático sería mediante una simple función a nivel de módulo:

```
def getcount():
    return C.count
```

Si has estructurado tu código para definir una clase única (o una jerarquía de clases altamente relacionadas) por módulo, esto proporcionará la encapsulación deseada.

2.6.9 ¿Como puedo sobrecargar constructores (o métodos) en Python?

Esta respuesta es aplicable, en realidad, a todos los métodos pero la pregunta suele surgir primero en el contexto de los constructores.

En C++ deberías escribir

```
class C {
    C() { cout << "No arguments\n"; }
    C(int i) { cout << "Argument is " << i << "\n"; }
}
```

En Python solo debes escribir un único constructor que tenga en cuenta todos los casos usando los argumentos por defecto. Por ejemplo:

```
class C:
    def __init__(self, i=None):
        if i is None:
            print("No arguments")
        else:
            print("Argument is", i)
```

Esto no es totalmente equivalente pero, en la práctica, es muy similar.

Podrías intentar, también una lista de argumentos de longitud variable, por ejemplo

```
def __init__(self, *args):
    ...
```

El mismo enfoque funciona para todas las definiciones de métodos.

2.6.10 Intento usar __spam y obtengo un error sobre _SomeClassName__spam.

Nombres de variable con doble guión prefijado se convierten, con una modificación de nombres, para proporcionar una forma simple pero efectiva de definir variables de clase privadas. Cualquier identificador de la forma `__spam` (como mínimo dos guiones bajos como prefijo, como máximo un guión bajo como sufijo) se reemplaza con `_classname__spam`, donde `classname` es el nombre de la clase eliminando cualquier guión bajo prefijado.

Esto no garantiza la privacidad: un usuario externo puede acceder, de forma deliberada y si así lo desea, al atributo «_classname__spam», y los valores privados son visibles en el `__dict__` del objeto. Muchos programadores Python no se suelen molestar en usar nombres privados de variables.

2.6.11 Mi clase define __del__ pero no se le invoca cuando borro el objeto.

Existen varias razones posibles para que suceda así.

La declaración del no invoca, necesariamente, al método `__del__()` – simplemente reduce el conteo de referencias del objeto y, si se reduce a cero entonces es cuando se invoca a `__del__()`.

Si tus estructuras de datos contienen enlaces circulares (por ejemplo un árbol en el cual cada hijo tiene una referencia al padre y cada padre tiene una lista de hijos) el conteo de referencias no alcanzará nunca el valor de cero. De vez en cuando, Python ejecuta un algoritmo para detectar esos ciclos pero el recolector de basura debe ejecutarse un rato después de que se desvanezca la última referencia a tu estructura de datos, de tal forma que tu método `__del__()` se pueda invocar en un momento aleatorio que no resulte inconveniente. Esto no es conveniente si estás intentando reproducir un problema. Peor aún, el orden en el que se ejecutan los métodos `__del__()` del objeto es arbitrario. Puedes ejecutar `gc.collect()` para forzar una recolección pero *existen* casos patológicos en los cuales los objetos nunca serán recolectados.

A pesar del recolector de ciclos, siempre será buena idea definir un método `close()` de forma explícita en objetos que debe ser llamado en el momento que has terminado con ellos. El método `close()` puede, en ese momento, eliminar atributos que se refieren a subobjetos. No invoques directamente a `__del__()` – `__del__()` debe invocar a `close()` y `close()` debe asegurarse que puede ser invocado más de una vez en el mismo objeto.

Otra forma de evitar referencias cíclicas sería usando el módulo `weakref`, que permite apuntar hacia objetos sin incrementar su conteo de referencias. Las estructuras de datos en árbol, por ejemplo, deberían usar referencias débiles para las referencias del padre y hermanos (¡si es que las necesitan!).

Finalmente, si tu método `__del__()` lanza una excepción, se manda un mensaje de alerta a `sys.stderr`.

2.6.12 ¿Cómo puedo obtener una lista de todas las instancias de una clase dada?

Python no hace seguimiento de todas las instancias de una clase (o de los tipos incorporados). Puedes programar el constructor de una clase para que haga seguimiento de todas sus instancias manteniendo una lista de referencias débiles a cada instancia.

2.6.13 ¿Por qué el resultado de `id()` no parece ser único?

La función incorporada `id()` devuelve un entero que se garantiza que sea único durante la vida del objeto. Debido a que en CPython esta es la dirección en memoria del objeto, sucede que, frecuentemente, después de que un objeto se elimina de la memoria el siguiente objeto recién creado se localiza en la misma posición en memoria. Esto se puede ver ilustrado en este ejemplo:

```
>>> id(1000)
13901272
>>> id(2000)
13901272
```

Las dos ids pertenecen a dos objetos "entero" diferentes que se crean antes y se eliminan inmediatamente después de la ejecución de la invocación a `id()`. Para estar seguro que los objetos cuya id quieres examinar siguen vivos crea otra referencia al objeto:

```
>>> a = 1000; b = 2000
>>> id(a)
13901272
>>> id(b)
13891296
```

2.7 Módulos

2.7.1 ¿Cómo creo un fichero .pyc?

Cuando se importa un módulo por primera vez (o cuando el código fuente ha cambiado desde que el fichero compilado se creó) un fichero `.pyc` que contiene el código compilado se debería crear en la subcarpeta `__pycache__` del directorio que contiene al fichero".py". El fichero `.pyc` tendrá un nombre que empezará con el mismo nombre que el del fichero `.py` y terminará con `.pyc`, con un componente intermedio que dependerá del binario `python` en particular que lo creó. (Ver **PEP 3147** para detalles.)

Una razón por la que no se cree un fichero `.pyc` podría ser debido a un problema de permisos del directorio que contiene al fichero fuente, lo que significa que el subdirectorio `__pycache__` no se puede crear. Esto puede suceder, por ejemplo, si desarrollas como un usuario pero lo ejecutas como otro, como si estuvieras probando en un servidor web.

Hasta que no definas la variable de entorno `PYTHONDONTWRITEBYTECODE`, la creación de un fichero .pyc se hará automáticamente si importas un módulo y Python dispone de la habilidad (permisos, espacio libre, etc…) para crear un subdirectorio `__pycache__` y escribir un módulo compilado en ese subdirectorio.

La ejecución de un script principal Python no se considera una importación y no se crea el fichero `.pyc`. Por ejemplo, Si tienes un módulo principal `foo.py` que importa a otro módulo `xyz.py`, cuando ejecutas `foo` (mediante un comando de la shell `python foo.py`), se creará un fichero `.pyc` para `xyz` porque `xyz` ha sido importado, pero no se creará un fichero `.pyc` para `foo` ya que `foo.py` no ha sido importado.

Si necesitas crear un fichero `.pyc` también para `foo` – es decir, crear un fichero `.pyc` para un módulo que no ha sido importado – puedes usar los módulos `py_compile` y `compileall`.

El módulo `py_compile` puede compilar manualmente cualquier módulo. Una forma sería usando la función `compile()` de ese módulo de forma interactiva:

```
>>> import py_compile
>>> py_compile.compile('foo.py')
```

Esto escribirá `.pyc` en el subdirectorio `__pycache__` en la misma localización en la que se encuentre `foo.py` (o, puedes sobreescribir ese comportamiento con el parámetro opcional `cfile`).

Puedes compilar automáticamente todos los ficheros en un directorio o directorios usando el módulo `compileall`. Lo puedes hacer desde la línea de comandos ejecutando `compileall.py` y proporcionando una ruta al directorio que contiene los ficheros Python a compilar:

```
python -m compileall .
```

2.7.2 ¿Cómo puedo encontrar el nombre del módulo en uso?

Un módulo puede encontrar su propio nombre mirando en la variable global predeterminada `__name__`. Si tiene el valor `'__main__'`, el programa se está ejecutando como un script. Muchos módulos que se usan, generalmente, importados en otro script proporcionan, además, una interfaz para la línea de comandos o para probarse a si mismos y solo ejecutan código después de comprobar `__name__`:

```
def main():
    print('Running test...')
    ...

if __name__ == '__main__':
    main()
```

2.7.3 ¿Cómo podría tener módulos que se importan mutuamente entre ellos?

Supón que tienes los siguientes módulos:

foo.py:

```
from bar import bar_var
foo_var = 1
```

bar.py:

```
from foo import foo_var
bar_var = 2
```

El problema es que el intérprete realizará los siguientes pasos:

- main importa a foo
- Se crea un *globals* vacío para foo
- foo se compila y se comienza a ejecutar
- foo importa a bar
- Se crea un *globals* vacío para bar
- bar se compila y se comienza a ejecutar
- bar importa a foo (lo cual no es una opción ya que ya hay un módulo que se llama foo)
- bar.foo_var = foo.foo_var

El último paso falla debido a que Python todavía no ha terminado de interpretar a `foo` y el diccionario de símbolos global para `foo` todavía se encuentra vacío.

Lo mismo ocurre cuando usas `import foo` y luego tratas de acceder a `foo.foo_var` en un código global.

Existen (al menos) tres posibles soluciones para este problema.

Guido van Rossum recomienda evitar todos los usos de `from <module> import ...`, y colocar todo el código dentro de funciones. La inicialización de variables globales y variables de clase debería usar únicamente constantes o funciones incorporadas . Esto significa que todo se referenciará como `<module>.<name>` desde un módulo importado.

Jim Roskind sugiere realizar los siguientes pasos en el siguiente orden en cada módulo:

- exportar (*globals*, funciones y clases que no necesitan clases bases importadas)
- `import` declaraciones
- código activo (incluyendo *globals* que han sido inicializados desde valores importados).

este enfoque no le gusta mucho a van Rossum debido a que los import aparecen en lugares extraños, pero funciona.

Matthias Urlichs recomienda reestructurar tu código de tal forma que un import recursivo no sea necesario.

Estas soluciones no son mutuamente excluyentes.

2.7.4 __import__("x.y.z") devuelve <module "x">; ¿cómo puedo obtener z?

Considera, en su lugar, usa la función de conveniencia `import_module()` de `importlib`:

```
z = importlib.import_module('x.y.z')
```

2.7.5 Cuando edito un módulo importado y lo reimporto los cambios no tienen efecto. ¿Por qué sucede esto?

Por razones de eficiencia además de por consistencia, Python solo lee el fichero del módulo la primera vez que el módulo se importa. Si no lo hiciera así, un programa escrito en muchos módulos donde cada módulo importa al mismo módulo básico estaría analizando sintácticamente el mismo módulo básico muchas veces. Para forzar una relectura de un módulo que ha sido modificado haz lo siguiente:

```
import importlib
import modname
importlib.reload(modname)
```

Alerta: esta técnica no es 100% segura. En particular, los módulos que contienen declaraciones como

```
from modname import some_objects
```

continuarán funcionando con la versión antigua de los objetos importados. Si el módulo contiene definiciones de clase, instancias de clase ya existentes *no* se actualizarán para usar la nueva definición de la clase. Esto podría resultar en el comportamiento paradójico siguiente:

```
>>> import importlib
>>> import cls
>>> c = cls.C()                # Create an instance of C
>>> importlib.reload(cls)
<module 'cls' from 'cls.py'>
>>> isinstance(c, cls.C)       # isinstance is false?!?
False
```

La naturaleza del problema se hace evidente si muestras la «identity» de los objetos clase:

```
>>> hex(id(c.__class__))
'0x7352a0'
>>> hex(id(cls.C))
'0x4198d0'
```

CAPÍTULO 3

Preguntas frecuentes sobre diseño e historia

3.1 ¿Por qué Python usa indentación para agrupar declaraciones?

Guido van Rossum cree que usar indentación para agrupar es extremadamente elegante y contribuye mucho a la claridad del programa Python promedio. La mayoría de las personas aprenden a amar esta característica después de un tiempo.

Como no hay corchetes de inicio/fin, no puede haber un desacuerdo entre la agrupación percibida por el analizador y el lector humano. Ocasionalmente, los programadores de C encontrarán un fragmento de código como este:

```
if (x <= y)
        x++;
        y--;
z++;
```

Solo se ejecuta la instrucción `x ++` si la condición es verdadera, pero la indentación lo lleva a creer lo contrario. Incluso los programadores experimentados de C a veces lo miran durante mucho tiempo preguntándose por qué `y` se está disminuyendo incluso para `x > y`.

Debido a que no hay corchetes de inicio/fin, Python es mucho menos propenso a conflictos de estilo de codificación. En C hay muchas formas diferentes de colocar las llaves. Si está acostumbrado a leer y escribir código que usa un estilo, se sentirá al menos un poco incómodo cuando lea (o le pidan que escriba) otro estilo.

Muchos estilos de codificación colocan corchetes de inicio / fin en una línea por sí mismos. Esto hace que los programas sean considerablemente más largos y desperdicia un valioso espacio en la pantalla, lo que dificulta obtener una buena visión general de un programa. Idealmente, una función debería caber en una pantalla (por ejemplo, 20-30 líneas). 20 líneas de Python pueden hacer mucho más trabajo que 20 líneas de C. Esto no se debe únicamente a la falta de corchetes de inicio/fin – la falta de declaraciones y los tipos de datos de alto nivel también son responsables – sino también la indentación basada en la sintaxis ciertamente ayuda.

3.2 ¿Por qué obtengo resultados extraños con operaciones aritméticas simples?

Ver la siguiente pregunta.

3.3 ¿Por qué los cálculos de punto flotante son tan inexactos?

Los usuarios a menudo se sorprenden por resultados como este:

```
>>> 1.2 - 1.0
0.19999999999999996
```

y creo que es un error en Python. No es. Esto tiene poco que ver con Python, y mucho más con la forma en que la plataforma subyacente maneja los números de punto flotante.

El tipo `float` en CPython usa una C `double` para el almacenamiento. Un valor del objeto `float` se almacena en coma flotante binaria con una precisión fija (típicamente 53 bits) y Python usa operaciones C, que a su vez dependen de la implementación de hardware en el procesador, para realizar operaciones de coma flotante. Esto significa que, en lo que respecta a las operaciones de punto flotante, Python se comporta como muchos lenguajes populares, incluidos C y Java.

Muchos números que se pueden escribir fácilmente en notación decimal no se pueden expresar exactamente en coma flotante binaria. Por ejemplo, después de:

```
>>> x = 1.2
```

el valor almacenado para `x` es una aproximación (muy buena) al valor decimal `1.2`, pero no es exactamente igual a él. En una máquina típica, el valor real almacenado es:

```
1.0011001100110011001100110011001100110011001100110011 (binary)
```

que es exactamente:

```
1.1999999999999999555910790149937383830547332763671875 (decimal)
```

La precisión típica de 53 bits proporciona flotantes Python con 15–16 dígitos decimales de precisión.

Para obtener una explicación más completa, consulte el capítulo aritmética de coma flotante en el tutorial de Python.

3.4 ¿Por qué las cadenas de caracteres de Python son inmutables?

Hay varias ventajas.

Una es el rendimiento: saber que una cadena es inmutable significa que podemos asignarle espacio en el momento de la creación, y los requisitos de almacenamiento son fijos e inmutables. Esta es también una de las razones para la distinción entre tuplas y listas.

Otra ventaja es que las cadenas en Python se consideran tan «elementales» como los números. Ninguna cantidad de actividad cambiará el valor 8 a otra cosa, y en Python, ninguna cantidad de actividad cambiará la cadena «ocho» a otra cosa.

3.5 ¿Por qué debe usarse "self" explícitamente en las definiciones y llamadas de métodos?

La idea fue tomada de Modula-3. Resulta ser muy útil, por una variedad de razones.

Primero, es más obvio que está utilizando un método o atributo de instancia en lugar de una variable local. Leer `self.x` o `self.meth()` deja absolutamente claro que se usa una variable de instancia o método incluso si no conoce la definición de clase de memoria. En C++, puede darse cuenta de la falta de una declaración de variable local (suponiendo que los globales son raros o fácilmente reconocibles) – pero en Python, no hay declaraciones de variables locales, por lo que debería buscar la definición de clase para estar seguro. Algunos estándares de codificación de C++ y Java requieren que los atributos de instancia tengan un prefijo `m_`, porque el ser explícito también es útil en esos lenguajes.

En segundo lugar, significa que no es necesaria una sintaxis especial si desea hacer referencia explícita o llamar al método desde una clase en particular. En C++, si desea usar un método de una clase base que se anula en una clase derivada, debe usar el operador `::` – en Python puede escribir `baseclass.methodname(self, <argument list>)`. Esto es particularmente útil para métodos `__init__()`, y en general en los casos en que un método de clase derivada quiere extender el método de clase base del mismo nombre y, por lo tanto, tiene que llamar al método de clase base de alguna manera.

Finalmente, para las variables de instancia se resuelve un problema sintáctico con la asignación: dado que las variables locales en Python son (¡por definición!) Aquellas variables a las que se asigna un valor en un cuerpo de función (y que no se declaran explícitamente como globales), tiene que haber una forma de decirle al intérprete que una asignación estaba destinada a asignar a una variable de instancia en lugar de a una variable local, y que preferiblemente debería ser sintáctica (por razones de eficiencia). C++ hace esto a través de declaraciones, pero Python no tiene declaraciones y sería una pena tener que presentarlas solo para este propósito. Usar el `self.var` explícito resuelve esto muy bien. Del mismo modo, para usar variables de instancia, tener que escribir `self.var` significa que las referencias a nombres no calificados dentro de un método no tienen que buscar en los directorios de la instancia. Para decirlo de otra manera, las variables locales y las variables de instancia viven en dos espacios de nombres diferentes, y debe decirle a Python qué espacio de nombres usar.

3.6 ¿Por qué no puedo usar una tarea en una expresión?

¡A partir de Python 3.8, se puede!

Asignación de expresiones usando el operador de morsa := asigna una variable en una expresión:

```
while chunk := fp.read(200):
   print(chunk)
```

Ver **PEP 572** para más información.

3.7 ¿Por qué Python usa métodos para alguna funcionalidad (por ejemplo, list.index()) pero funciones para otra (por ejemplo, len(list))?

Como dijo Guido:

> (a) Para algunas operaciones, la notación de prefijo solo se lee mejor que postfijo – las operaciones de prefijo (e ¡infijo!) tienen una larga tradición en matemáticas que le gustan las anotaciones donde las imágenes ayudan al matemático a pensar en un problema. Compare lo fácil con que reescribimos una fórmula como x*(a+b) en x*a + x*b con la torpeza de hacer lo mismo usando una notación OO sin procesar.
>
> (b) Cuando leo un código que dice len(x), *sé* que está pidiendo la longitud de algo. Esto me dice dos cosas: el resultado es un número entero y el argumento es algún tipo de contenedor. Por el contrario,

> cuando leo x.len (), ya debo saber que x es algún tipo de contenedor que implementa una interfaz o hereda de una clase que tiene un estándar len(). Sea testigo de la confusión que ocasionalmente tenemos cuando una clase que no está implementando una asignación tiene un método get() o keys(), o algo que no es un archivo tiene un método write().
>
> —https://mail.python.org/pipermail/python-3000/2006-November/004643.html

3.8 ¿Por qué join() es un método de cadena de caracteres en lugar de un método de lista o tupla?

Las cadenas de caracteres se volvieron mucho más parecidas a otros tipos estándar a partir de Python 1.6, cuando se agregaron métodos que brindan la misma funcionalidad que siempre ha estado disponible utilizando las funciones del módulo de cadenas. La mayoría de estos nuevos métodos han sido ampliamente aceptados, pero el que parece hacer que algunos programadores se sientan incómodos es:

```
", ".join(['1', '2', '4', '8', '16'])
```

que da el resultado:

```
"1, 2, 4, 8, 16"
```

Hay dos argumentos comunes en contra de este uso.

El primero corre a lo largo de las líneas de: «Se ve realmente feo el uso de un método de un literal de cadena (constante de cadena)», a lo que la respuesta es que sí, pero un literal de cadena es solo un valor fijo. Si se permiten los métodos en nombres vinculados a cadenas, no hay razón lógica para que no estén disponibles en literales.

La segunda objeción generalmente se presenta como: «Realmente estoy diciéndole a una secuencia que una a sus miembros junto con una constante de cadena». Lamentablemente, no lo estas haciendo. Por alguna razón, parece ser mucho menos difícil tener `split()` como método de cadena, ya que en ese caso es fácil ver que:

```
"1, 2, 4, 8, 16".split(", ")
```

es una instrucción a un literal de cadena para retornar las subcadenas de caracteres delimitadas por el separador dado (o, por defecto, ejecuciones arbitrarias de espacio en blanco).

`join()` es un método de cadena de caracteres porque al usarlo le está diciendo a la cadena de separación que itere sobre una secuencia de cadenas y se inserte entre elementos adyacentes. Este método se puede usar con cualquier argumento que obedezca las reglas para los objetos de secuencia, incluidas las clases nuevas que pueda definir usted mismo. Existen métodos similares para bytes y objetos bytearray.

3.9 ¿Qué tan rápido van las excepciones?

Un bloque try/except es extremadamente eficiente si no se generan excepciones. En realidad, capturar una excepción es costoso. En versiones de Python anteriores a la 2.0, era común usar este modismo:

```
try:
    value = mydict[key]
except KeyError:
    mydict[key] = getvalue(key)
    value = mydict[key]
```

Esto solo tenía sentido cuando esperaba que el dict tuviera la clave casi todo el tiempo. Si ese no fuera el caso, lo codificó así:

```
if key in mydict:
    value = mydict[key]
else:
    value = mydict[key] = getvalue(key)
```

Para este caso específico, también podría usar `value = dict.setdefault(key, getvalue(key))`, pero solo si la llamada `getvalue()` es lo suficientemente barata porque se evalúa en todos los casos.

3.10 ¿Por qué no hay un *switch* o una declaración *case* en Python?

Puede hacer esto fácilmente con una secuencia de `if... elif... elif... else`. Ha habido algunas propuestas para cambiar la sintaxis de la declaración, pero todavía no hay consenso sobre si y cómo hacer pruebas de rango. Ver **PEP 275** para detalles completos y el estado actual.

Para los casos en los que necesita elegir entre una gran cantidad de posibilidades, puede crear un diccionario que asigne valores de casos a funciones para llamar. Por ejemplo:

```
def function_1(...):
    ...

functions = {'a': function_1,
             'b': function_2,
             'c': self.method_1, ...}

func = functions[value]
func()
```

Para invocar métodos en objetos, puede simplificar aún más utilizando `getattr()` incorporado para recuperar métodos con un nombre particular:

```
def visit_a(self, ...):
    ...
...

def dispatch(self, value):
    method_name = 'visit_' + str(value)
    method = getattr(self, method_name)
    method()
```

Se sugiere que utilice un prefijo para los nombres de los métodos, como `visit_` en este ejemplo. Sin dicho prefijo, si los valores provienen de una fuente no confiable, un atacante podría invocar cualquier método en su objeto.

3.11 ¿No puede emular hilos en el intérprete en lugar de confiar en una implementación de hilos específica del sistema operativo?

Respuesta 1: Desafortunadamente, el intérprete empuja al menos un marco de pila C para cada marco de pila de Python. Además, las extensiones pueden volver a llamar a Python en momentos casi aleatorios. Por lo tanto, una implementación completa de subprocesos requiere soporte de subprocesos para C.

Respuesta 2: Afortunadamente, existe Python sin pila, que tiene un bucle de intérprete completamente rediseñado que evita la pila C.

3.12 ¿Por qué las expresiones lambda no pueden contener sentencias?

Las expresiones lambda de Python no pueden contener declaraciones porque el marco sintáctico de Python no puede manejar declaraciones anidadas dentro de expresiones. Sin embargo, en Python, este no es un problema grave. A diferencia de las formas lambda en otros lenguajes, donde agregan funcionalidad, las lambdas de Python son solo una notación abreviada si eres demasiado vago para definir una función.

Las funciones ya son objetos de primera clase en Python y pueden declararse en un ámbito local. Por lo tanto, la única ventaja de usar una lambda en lugar de una función definida localmente es que no es necesario inventar un nombre para la función, sino que es solo una variable local para la cual el objeto de función (que es exactamente el mismo tipo de se asigna un objeto que produce una expresión lambda)

3.13 ¿Se puede compilar Python en código máquina, C o algún otro lenguaje?

Cython compila una versión modificada de Python con anotaciones opcionales en extensiones C. Nuitka es un compilador prometedor de Python en código C ++, con el objetivo de soportar el lenguaje completo de Python. Para compilar en Java puede considerar VOC.

3.14 ¿Cómo gestiona Python la memoria?

Los detalles de la administración de memoria de Python dependen de la implementación. La implementación estándar de Python, *CPython*, utiliza el recuento de referencias para detectar objetos inaccesibles, y otro mecanismo para recopilar ciclos de referencia, ejecutando periódicamente un algoritmo de detección de ciclos que busca ciclos inaccesibles y elimina los objetos involucrados. El módulo `gc` proporciona funciones para realizar una recolección de basura, obtener estadísticas de depuración y ajustar los parámetros del recolector.

Sin embargo, otras implementaciones (como Jython o PyPy) pueden confiar en un mecanismo diferente, como recolector de basura. Esta diferencia puede causar algunos problemas sutiles de portabilidad si su código de Python depende del comportamiento de la implementación de conteo de referencias.

En algunas implementaciones de Python, el siguiente código (que está bien en CPython) probablemente se quedará sin descriptores de archivo:

```
for file in very_long_list_of_files:
    f = open(file)
    c = f.read(1)
```

De hecho, utilizando el esquema de conteo de referencias y destructor de CPython, cada nueva asignación a *f* cierra el archivo anterior. Sin embargo, con un GC tradicional, esos objetos de archivo solo se recopilarán (y cerrarán) a intervalos variables y posiblemente largos.

Si desea escribir código que funcione con cualquier implementación de Python, debe cerrar explícitamente el archivo o utilizar una declaración `with`; esto funcionará independientemente del esquema de administración de memoria:

```
for file in very_long_list_of_files:
    with open(file) as f:
        c = f.read(1)
```

3.15 ¿Por qué CPython no utiliza un esquema de recolección de basura más tradicional?

Por un lado, esta no es una característica estándar de C y, por lo tanto, no es portátil. (Sí, sabemos acerca de la biblioteca Boehm GC. Tiene fragmentos de código de ensamblador para *la mayoría* de las plataformas comunes, no para todas ellas, y aunque es principalmente transparente, no es completamente transparente; se requieren parches para obtener Python para trabajar con eso)

El GC tradicional también se convierte en un problema cuando Python está integrado en otras aplicaciones. Mientras que en un Python independiente está bien reemplazar el estándar malloc() y free() con versiones proporcionadas por la biblioteca GC, una aplicación que incruste Python puede querer tener su *propio* sustituto de malloc() y free(), y puede No quiero a Python. En este momento, CPython funciona con todo lo que implementa malloc() y free() correctamente.

3.16 ¿Por qué no se libera toda la memoria cuando sale CPython?

Los objetos a los que se hace referencia desde los espacios de nombres globales de los módulos de Python no siempre se desasignan cuando Python sale. Esto puede suceder si hay referencias circulares. También hay ciertos bits de memoria asignados por la biblioteca de C que son imposibles de liberar (por ejemplo, una herramienta como Purify se quejará de estos). Python es, sin embargo, agresivo sobre la limpieza de la memoria al salir e intenta destruir cada objeto.

Si desea forzar a Python a eliminar ciertas cosas en la desasignación, use el módulo `atexit` para ejecutar una función que obligará a esas eliminaciones.

3.17 ¿Por qué hay tipos de datos separados de tuplas y listas?

Las listas y las tuplas, si bien son similares en muchos aspectos, generalmente se usan de maneras fundamentalmente diferentes. Se puede pensar que las tuplas son similares a los registros Pascal o estructuras C; son pequeñas colecciones de datos relacionados que pueden ser de diferentes tipos que funcionan como un grupo. Por ejemplo, una coordenada cartesiana se representa adecuadamente como una tupla de dos o tres números.

Las listas, por otro lado, son más como matrices en otros lenguajes. Tienden a contener un número variable de objetos, todos los cuales tienen el mismo tipo y que se operan uno por uno. Por ejemplo, `os.listdir('.')` Retorna una lista de cadenas de caracteres que representan los archivos en el directorio actual. Las funciones que operan en esta salida generalmente no se romperían si agregara otro archivo o dos al directorio.

Las tuplas son inmutables, lo que significa que una vez que se ha creado una tupla, no puede reemplazar ninguno de sus elementos con un nuevo valor. Las listas son mutables, lo que significa que siempre puede cambiar los elementos de una lista. Solo los elementos inmutables se pueden usar como claves de diccionario y, por lo tanto, solo las tuplas y no las listas se pueden usar como claves.

3.18 ¿Cómo se implementan las listas en Python?

Las listas de CPython son realmente matrices de longitud variable, no listas enlazadas al estilo Lisp. La implementación utiliza una matriz contigua de referencias a otros objetos y mantiene un puntero a esta matriz y la longitud de la matriz en una estructura de encabezado de lista.

Esto hace que indexar una lista `a[i]` una operación cuyo costo es independiente del tamaño de la lista o del valor del índice.

Cuando se añaden o insertan elementos, la matriz de referencias cambia de tamaño. Se aplica cierta inteligencia para mejorar el rendimiento de la adición de elementos repetidamente; cuando la matriz debe crecer, se asigna un espacio extra para que las próximas veces no requieran un cambio de tamaño real.

3.19 ¿Cómo se implementan los diccionarios en CPython?

Los diccionarios de CPython se implementan como tablas hash redimensionables. En comparación con los árboles B (*B-trees*), esto proporciona un mejor rendimiento para la búsqueda (la operación más común con diferencia) en la mayoría de las circunstancias, y la implementación es más simple.

Los diccionarios funcionan calculando un código hash para cada clave almacenada en el diccionario utilizando la función incorporada `hash()`. El código hash varía ampliamente según la clave y una semilla por proceso; por ejemplo, «Python» podría dividir en hash a -539294296 mientras que «python», una cadena que difiere en un solo bit, podría dividir en 1142331976. El código de resumen se usa para calcular una ubicación en una matriz interna donde se almacenará el valor . Suponiendo que está almacenando claves que tienen valores hash diferentes, esto significa que los diccionarios toman tiempo constante – O(1), en notación Big-O – para recuperar una clave.

3.20 ¿Por qué las claves del diccionario deben ser inmutables?

La implementación de la tabla hash de los diccionarios utiliza un valor hash calculado a partir del valor clave para encontrar la clave. Si la clave fuera un objeto mutable, su valor podría cambiar y, por lo tanto, su hash también podría cambiar. Pero dado que quien cambie el objeto clave no puede decir que se estaba utilizando como clave de diccionario, no puede mover la entrada en el diccionario. Luego, cuando intente buscar el mismo objeto en el diccionario, no se encontrará porque su valor hash es diferente. Si trató de buscar el valor anterior, tampoco lo encontraría, porque el valor del objeto que se encuentra en ese hash bin sería diferente.

Si desea un diccionario indexado con una lista, simplemente convierta la lista a una tupla primero; La función `tuple(L)` crea una tupla con las mismas entradas que la lista `L`. Las tuplas son inmutables y, por lo tanto, pueden usarse como claves de diccionario.

Algunas soluciones inaceptables que se han propuesto:

- Listas de hash por su dirección (ID de objeto). Esto no funciona porque si construye una nueva lista con el mismo valor, no se encontrará; por ejemplo:

```
mydict = {[1, 2]: '12'}
print(mydict[[1, 2]])
```

  generaría una excepción `KeyError` porque la identificación del `[1, 2]` usado en la segunda línea difiere de la de la primera línea. En otras palabras, las claves del diccionario deben compararse usando ==, no usando `is`.

- Hacer una copia cuando use una lista como clave. Esto no funciona porque la lista, al ser un objeto mutable, podría contener una referencia a sí misma, y luego el código de copia se ejecutaría en un bucle infinito.

- Permitir listas como claves pero decirle al usuario que no las modifique. Esto permitiría una clase de errores difíciles de rastrear en los programas cuando olvidó o modificó una lista por accidente. También invalida una invariante importante de diccionarios: cada valor en `d.keys()` se puede usar como una clave del diccionario.

- Marcar las listas como de solo lectura una vez que se usan como clave de diccionario. El problema es que no solo el objeto de nivel superior puede cambiar su valor; podría usar una tupla que contiene una lista como clave. Ingresar cualquier cosa como clave en un diccionario requeriría marcar todos los objetos accesibles desde allí como de solo lectura – y nuevamente, los objetos autoreferenciados podrían causar un bucle infinito.

Hay un truco para evitar esto si lo necesita, pero úselo bajo su propio riesgo: puede envolver una estructura mutable dentro de una instancia de clase que tenga un método `__eq__()` y a `__hash__()` . Luego debe asegurarse de que el valor hash para todos los objetos de contenedor que residen en un diccionario (u otra estructura basada en hash) permanezca fijo mientras el objeto está en el diccionario (u otra estructura).

```
class ListWrapper:
    def __init__(self, the_list):
        self.the_list = the_list

    def __eq__(self, other):
```

(continué en la próxima página)

(proviene de la página anterior)

```
        return self.the_list == other.the_list

    def __hash__(self):
        l = self.the_list
        result = 98767 - len(l)*555
        for i, el in enumerate(l):
            try:
                result = result + (hash(el) % 9999999) * 1001 + i
            except Exception:
                result = (result % 7777777) + i * 333
        return result
```

Tenga en cuenta que el cálculo de hash se complica por la posibilidad de que algunos miembros de la lista sean inquebrantables y también por la posibilidad de desbordamiento aritmético.

Además, siempre debe darse el caso de que si `o1 == o2` (es decir, `o1.__eq__(o2) is True`), entonces `hash(o1) == hash(o2)` (es decir, `o1.__hash__() == o2.__hash__()`), independientemente de si el objeto está en un diccionario o no. Si no cumple con estas restricciones, los diccionarios y otras estructuras basadas en hash se comportarán mal.

En el caso de ListWrapper, siempre que el objeto contenedor esté en un diccionario, la lista ajustada no debe cambiar para evitar anomalías. No haga esto a menos que esté preparado para pensar detenidamente sobre los requisitos y las consecuencias de no cumplirlos correctamente. Considérese advertido.

3.21 ¿Por qué list.sort() no retorna la lista ordenada?

En situaciones donde el rendimiento es importante, hacer una copia de la lista solo para ordenarlo sería un desperdicio. Por lo tanto, `list.sort()` ordena la lista en su lugar. Para recordarle ese hecho, no retorna la lista ordenada. De esta manera, no se dejará engañar por sobreescribir accidentalmente una lista cuando necesite una copia ordenada, pero también deberá mantener la versión sin ordenar.

Si desea retornar una nueva lista, use la función incorporada `sorted()` en su lugar. Esta función crea una nueva lista a partir de un iterativo proporcionado, la ordena y la retorna. Por ejemplo, a continuación se explica cómo iterar sobre las teclas de un diccionario en orden ordenado:

```
for key in sorted(mydict):
    ...  # do whatever with mydict[key]...
```

3.22 ¿Cómo se especifica y aplica una especificación de interfaz en Python?

Una especificación de interfaz para un módulo proporcionada por lenguajes como C++ y Java describe los prototipos para los métodos y funciones del módulo. Muchos sienten que la aplicación en tiempo de compilación de las especificaciones de la interfaz ayuda en la construcción de grandes programas.

Python 2.6 agrega un módulo `abc` que le permite definir clases base abstractas (ABC). Luego puede usar `isinstance()` y `issubclass()` para verificar si una instancia o una clase implementa un ABC en particular. El módulo `collections.abc` define un conjunto de ABC útiles como `Iterable`, `Container` y `MutableMapping`.

For Python, many of the advantages of interface specifications can be obtained by an appropriate test discipline for components.

Un buen conjunto de pruebas para un módulo puede proporcionar una prueba de regresión y servir como una especificación de interfaz de módulo y un conjunto de ejemplos. Muchos módulos de Python se pueden ejecutar como un script para proporcionar una simple «autocomprobación». Incluso los módulos que usan interfaces externas complejas a menudo se pueden probar de forma aislada utilizando emulaciones triviales de «stub» de la interfaz externa.

Los módulos `doctest` y `unittest` o marcos de prueba de terceros se pueden utilizar para construir conjuntos de pruebas exhaustivas que ejercitan cada línea de código en un módulo.

Una disciplina de prueba adecuada puede ayudar a construir grandes aplicaciones complejas en Python, así como tener especificaciones de interfaz. De hecho, puede ser mejor porque una especificación de interfaz no puede probar ciertas propiedades de un programa. Por ejemplo, se espera que el método `append()` agregue nuevos elementos al final de alguna lista interna; una especificación de interfaz no puede probar que su implementación `append()` realmente haga esto correctamente, pero es trivial verificar esta propiedad en un conjunto de pruebas.

Escribir conjuntos de pruebas es muy útil, y es posible que desee diseñar su código con miras a que sea fácilmente probado. Una técnica cada vez más popular, el desarrollo dirigido por pruebas, requiere escribir partes del conjunto de pruebas primero, antes de escribir el código real. Por supuesto, Python te permite ser descuidado y no escribir casos de prueba.

3.23 ¿Por qué no hay goto?

In the 1970s people realized that unrestricted goto could lead to messy «spaghetti» code that was hard to understand and revise. In a high-level language, it is also unneeded as long as there are ways to branch (in Python, with `if` statements and `or`, `and`, and `if-else` expressions) and loop (with `while` and `for` statements, possibly containing `continue` and `break`).

One can also use exceptions to provide a «structured goto» that works even across function calls. Many feel that exceptions can conveniently emulate all reasonable uses of the «go» or «goto» constructs of C, Fortran, and other languages. For example:

```
class label(Exception): pass  # declare a label

try:
    ...
    if condition: raise label()  # goto label
    ...
except label:  # where to goto
    pass
...
```

Esto no le permite saltar a la mitad de un bucle, pero de todos modos eso generalmente se considera un abuso de goto. Utilizar con moderación.

3.24 ¿Por qué las cadenas de caracteres sin formato (r-strings) no pueden terminar con una barra diagonal inversa?

Más precisamente, no pueden terminar con un número impar de barras invertidas: la barra invertida no emparejada al final escapa el carácter de comillas de cierre, dejando una cadena sin terminar.

Las cadenas de caracteres sin formato se diseñaron para facilitar la creación de entradas para procesadores (principalmente motores de expresión regular) que desean realizar su propio procesamiento de escape de barra invertida. Tales procesadores consideran que una barra invertida sin par es un error de todos modos, por lo que las cadenas de caracteres sin procesar no lo permiten. A cambio, le permiten pasar el carácter de comillas de cadena escapándolo con una barra invertida. Estas reglas funcionan bien cuando las cadenas de caracteres r (*r-strings*) se usan para el propósito previsto.

Si está intentando construir nombres de ruta de Windows, tenga en cuenta que todas las llamadas al sistema de Windows también aceptan barras diagonales:

```
f = open("/mydir/file.txt")  # works fine!
```

Si está tratando de construir una ruta para un comando de DOS, intente por ejemplo uno de los siguientes:

```
dir = r"\this\is\my\dos\dir" "\\"
dir = r"\this\is\my\dos\dir\ "[:-1]
dir = "\\this\\is\\my\\dos\\dir\\"
```

3.25 ¿Por qué Python no tiene una declaración «with» para las asignaciones de atributos?

Python tiene una declaración "with" que envuelve la ejecución de un bloque, llamando al código en la entrada y salida del bloque. Algunos lenguajes tienen una construcción que se ve así:

```
with obj:
    a = 1               # equivalent to obj.a = 1
    total = total + 1   # obj.total = obj.total + 1
```

En Python, tal construcción sería ambigua.

Otros lenguajes, como Object Pascal, Delphi y C ++, utilizan tipos estáticos, por lo que es posible saber, de manera inequívoca, a qué miembro se le está asignando. Este es el punto principal de la escritura estática: el compilador *siempre* conoce el alcance de cada variable en tiempo de compilación.

Python usa tipos dinámicos. Es imposible saber de antemano a qué atributo se hará referencia en tiempo de ejecución. Los atributos de los miembros pueden agregarse o eliminarse de los objetos sobre la marcha. Esto hace que sea imposible saber, a partir de una simple lectura, a qué atributo se hace referencia: ¿uno local, uno global o un atributo miembro?

Por ejemplo, tome el siguiente fragmento incompleto:

```
def foo(a):
    with a:
        print(x)
```

El fragmento supone que «a» debe tener un atributo miembro llamado «x». Sin embargo, no hay nada en Python que le diga esto al intérprete. ¿Qué debería suceder si «a» es, digamos, un número entero? Si hay una variable global llamada «x», ¿se usará dentro del bloque with? Como puede ver, la naturaleza dinámica de Python hace que tales elecciones sean mucho más difíciles.

Sin embargo, el beneficio principal de «with» y características de lenguaje similares (reducción del volumen del código) se puede lograr fácilmente en Python mediante la asignación. En vez de:

```
function(args).mydict[index][index].a = 21
function(args).mydict[index][index].b = 42
function(args).mydict[index][index].c = 63
```

escribe esto:

```
ref = function(args).mydict[index][index]
ref.a = 21
ref.b = 42
ref.c = 63
```

Esto también tiene el efecto secundario de aumentar la velocidad de ejecución porque los enlaces de nombres se resuelven en tiempo de ejecución en Python, y la segunda versión solo necesita realizar la resolución una vez.

3.26 ¿Por qué se requieren dos puntos para las declaraciones if/while/def/class?

Los dos puntos se requieren principalmente para mejorar la legibilidad (uno de los resultados del lenguaje ABC experimental). Considera esto:

```
if a == b
    print(a)
```

versus

```
if a == b:
    print(a)
```

Observe cómo el segundo es un poco más fácil de leer. Observe más a fondo cómo los dos puntos establecen el ejemplo en esta respuesta de preguntas frecuentes; Es un uso estándar en inglés.

Otra razón menor es que los dos puntos facilitan a los editores con resaltado de sintaxis; pueden buscar dos puntos para decidir cuándo se debe aumentar la indentación en lugar de tener que hacer un análisis más elaborado del texto del programa.

3.27 ¿Por qué Python permite comas al final de las listas y tuplas?

Python le permite agregar una coma final al final de las listas, tuplas y diccionarios:

```
[1, 2, 3,]
('a', 'b', 'c',)
d = {
    "A": [1, 5],
    "B": [6, 7],  # last trailing comma is optional but good style
}
```

Hay varias razones para permitir esto.

Cuando tiene un valor literal para una lista, tupla o diccionario distribuido en varias líneas, es más fácil agregar más elementos porque no tiene que recordar agregar una coma a la línea anterior. Las líneas también se pueden reordenar sin crear un error de sintaxis.

La omisión accidental de la coma puede ocasionar errores difíciles de diagnosticar. Por ejemplo:

```
x = [
  "fee",
  "fie"
  "foo",
  "fum"
]
```

Parece que esta lista tiene cuatro elementos, pero en realidad contiene tres: «fee», «fiefoo» y «fum». Agregar siempre la coma evita esta fuente de error.

Permitir la coma final también puede facilitar la generación de código programático.

CAPÍTULO 4

Preguntas frecuentes sobre bibliotecas y extensiones

4.1 Cuestiones generales sobre bibliotecas

4.1.1 ¿Cómo encuentro un módulo o aplicación para ejecutar la tarea X?

Vea la referencia de bibliotecas para comprobar si existe un módulo relevante en la biblioteca estándar. (Eventualmente aprenderá lo que hay en la biblioteca estándar y será capaz de saltarse este paso.)

Para los paquetes de terceros, busque en el Índice de Paquetes de Python o pruebe Google u otro motor de búsqueda. Buscando por «Python» más una o dos palabras clave del tema de interés, generalmente encontrará algo útil.

4.1.2 ¿Dónde está el fichero fuente *math.py* (*socket.py*, *regex.py*, etc.)?

Si no puede encontrar un fichero fuente para un módulo, puede ser un módulo incorporado o cargado dinámicamente implementado en C, C++ u otro lenguaje compilado. En este caso puede no disponer del fichero fuente o puede ser algo como `mathmodule.c`, en algún lugar de un directorio fuente C (fuera del Python *Path*).

Hay (al menos) tres tipos de módulos en Python:

1) módulos escritos en Python (.py);
2) módulos escritos en C y cargados dinámicamente (*.dll, .pyd, .so, .sl*, etc.);
3) módulos escritos en C y enlazados con el intérprete; para obtener una lista de estos, escriba:

```
import sys
print(sys.builtin_module_names)
```

4.1.3 ¿Cómo hago ejecutable un script Python en Unix?

Necesita hacer dos cosas: el modo del fichero del script debe ser ejecutable y la primera línea debe comenzar con # ! seguido de la ruta al intérprete de Python.

Lo primero se hace ejecutando `chmod +x scriptfile` o bien `chmod 755 scriptfile`.

Lo segundo se puede hacer de distintas maneras. La manera más directa es escribir

```
#!/usr/local/bin/python
```

en la primera línea de su fichero, usando la ruta donde está instalado el intérprete de Python en su plataforma.

Si quiere que el script sea independiente de donde se ubique el intérprete de Python, puede usar el programa **env**. Casi todas las variantes de Unix soportan lo siguiente, asumiendo que el intérprete de Python está en un directorio del `PATH` de usuario:

```
#!/usr/bin/env python
```

No haga esto para scripts CGI. La variable `PATH` para scripts CGI es mínima, así que necesita usar la ruta real absoluta al intérprete.

Ocasionalmente, un entorno de usuario está tan lleno que el programa **/usr/bin/env** falla; o bien no existe el programa env. En ese caso, puede intentar el siguiente truco (gracias a Alex Rezinsky):

```
#! /bin/sh
""":"
exec python $0 ${1+"$@"}
"""
```

Una pequeña desventaja es que esto define el *__doc__* del script. Sin embargo, puede arreglarlo añadiendo

```
__doc__ = """...Whatever..."""
```

4.1.4 ¿Hay un paquete curses/termcap para Python?

Para variantes Unix: La distribución estándar de Python viene con un módulo curses en el subdirectorio Modules, aunque no está compilado por defecto. (Nótese que esto no está disponible en la distribución Windows — no hay módulo curses para Windows.)

El módulo `curses` soporta características básicas de cursores así como muchas funciones adicionales de ncurses y cursores SYSV como color, soporte para conjuntos de caracteres alternativos, pads, y soporte para ratón. Esto significa que el módulo no es compatible con sistemas operativos que sólo tienen cursores BSD, pero no parece que ningún sistema operativo actualmente mantenido caiga dentro de esta categoría.

Para Windows: use el módulo consolelib.

4.1.5 ¿Hay un equivalente en Python al onexit() de C?

El módulo `atexit` proporciona una función de registro que es similar a `onexit()` de C.

4.1.6 ¿Por qué no funcionan mis manejadores de señales?

El problema más común es que el manejador de señales esté declarado con la lista incorrecta de argumentos. Se llama como

```
handler(signum, frame)
```

así que debería declararse con dos argumentos:

```
def handler(signum, frame):
    ...
```

4.2 Tareas comunes

4.2.1 ¿Cómo pruebo un programa o un componente Python?

Python viene con dos *frameworks* de *testing*. El módulo `doctest` encuentra ejemplos en los docstrings para un módulo y los ejecuta, comparando la salida con la salida esperada especificada en la cadena de documentación.

El módulo `unittest` es un *framework* de *testing* más agradable y modelado sobre los *frameworks* de *testing* de Java y Smalltalk.

Para hacer más fácil el *testing*, debería usar un buen diseño modular en su programa. Su programa debería tener casi toda la funcionalidad encapsulada en funciones o en métodos de clases — y esto algunas veces tiene el efecto sorprendente y encantador de que su programa funcione más rápido (porque los accesos a las variables locales son más rápidas que los accesos a las variables globales). Además el programa debería evitar depender de la mutación de variables globales, ya que esto dificulta mucho más hacer el *testing*.

La «lógica global principal» de su programa puede ser tan simple como

```
if __name__ == "__main__":
    main_logic()
```

al final del módulo principal de su programa.

Una vez que su programa esté organizado en una colección manejable de funciones y comportamientos de clases, usted debería escribir funciones de comprobación que ejerciten los comportamientos. Se puede asociar un conjunto de pruebas a cada módulo. Esto suena a mucho trabajo, pero gracias a que Python es tan conciso y flexible, se hace sorprendentemente fácil. Puede codificar de manera mucho más agradable y divertida escribiendo funciones de comprobación en paralelo con el «código de producción», ya que esto facilita encontrar antes errores e incluso fallos de diseño.

Los «módulos de soporte» que no tienen la intención de estar en el módulo principal de un programa pueden incluir un auto *test* del módulo.

```
if __name__ == "__main__":
    self_test()
```

Incluso los programas que interactúan con interfaces externas complejas se pueden comprobar cuando las interfaces externas no están disponibles usando interfaces «simuladas» implementadas en Python.

4.2.2 ¿Cómo creo documentación a partir de los docstrings?

El módulo `pydoc` puede crear HTML desde los docstrings existentes en su código fuente Python. Una alternativa para crear documentación API estrictamente desde docstrings es epydoc. Sphinx también puede incluir contenido docstring.

4.2.3 ¿Cómo consigo presionar una única tecla cada vez?

Para variantes Unix hay varias soluciones. Lo más directo es hacerlo usando cursores, pero curses es un módulo bastante amplio para aprenderlo.

4.3 Hilos

4.3.1 ¿Cómo programo usando hilos?

Asegúrese de usar el módulo `threading` y no el módulo `_thread`. El módulo `threading` construye abstracciones convenientes sobre las primitivas de bajo nivel proporcionadas por el módulo `_thread`.

Aahz tiene un conjunto de transparencias en su tutorial de hilos que resulta útil: vea http://www.pythoncraft.com/OSCON2001/.

4.3.2 Ninguno de mis hilos parece funcionar: ¿por qué?

Tan pronto como el hilo principal termine, se matan todos los hilos. Su hilo principal está corriendo demasiado rápido, sin dar tiempo a los hilos para hacer algún trabajo.

Una solución sencilla es añadir un *sleep* al final del programa que sea suficientemente largo para que todos los hilos terminen:

```
import threading, time

def thread_task(name, n):
    for i in range(n):
        print(name, i)

for i in range(10):
    T = threading.Thread(target=thread_task, args=(str(i), i))
    T.start()

time.sleep(10)  # <---------------------------!
```

Por ahora (en muchas plataformas) los hilos no corren en paralelo, sino que parece que corren secuencialmente, ¡uno a la vez! La razón es que el planificador de hilos del sistema operativo no inicia un nuevo hilo hasta que el hilo anterior está bloqueado.

Una solución sencilla es añadir un pequeño *sleep* al comienzo de la función run:

```
def thread_task(name, n):
    time.sleep(0.001)  # <--------------------!
    for i in range(n):
        print(name, i)

for i in range(10):
    T = threading.Thread(target=thread_task, args=(str(i), i))
    T.start()

time.sleep(10)
```

En vez de intentar adivinar un valor de retardo adecuado para `time.sleep()`, es mejor usar algún tipo de mecanismo de semáforo. Una idea es usar el módulo `queue` para crear un objeto cola, permitiendo que cada hilo añada un *token* a la cola cuando termine, y permitiendo al hilo principal leer tantos tokens de la cola como hilos haya.

4.3.3 ¿Cómo puedo dividir trabajo entre un grupo de hilos?

La manera más fácil es usar el nuevo módulo `concurrent.futures`, especialmente la clase `ThreadPoolExecutor`.

O, si quiere tener un control más preciso sobre el algoritmo de despacho, puede escribir su propia lógica manualmente. Use el módulo `queue` para crear una cola que contenga una lista de trabajos. La clase `Queue` mantiene una lista de objetos y tiene un método `.put(obj)` que añade elementos a la cola y un método `.get()` que los retorna. Esta clase se encargará de los bloqueos necesarios para asegurar que cada trabajo se reparte exactamente una vez.

Aquí hay un ejemplo trivial:

```
import threading, queue, time

# The worker thread gets jobs off the queue.  When the queue is empty, it
# assumes there will be no more work and exits.
# (Realistically workers will run until terminated.)
def worker():
    print('Running worker')
    time.sleep(0.1)
    while True:
        try:
            arg = q.get(block=False)
        except queue.Empty:
            print('Worker', threading.currentThread(), end=' ')
            print('queue empty')
            break
        else:
            print('Worker', threading.currentThread(), end=' ')
            print('running with argument', arg)
            time.sleep(0.5)

# Create queue
q = queue.Queue()

# Start a pool of 5 workers
for i in range(5):
    t = threading.Thread(target=worker, name='worker %i' % (i+1))
    t.start()

# Begin adding work to the queue
for i in range(50):
    q.put(i)

# Give threads time to run
print('Main thread sleeping')
time.sleep(5)
```

Cuando se ejecute, esto producirá la siguiente salida:

```
Running worker
Running worker
Running worker
Running worker
Running worker
Main thread sleeping
Worker <Thread(worker 1, started 130283832797456)> running with argument 0
```

(continué en la próxima página)

(proviene de la página anterior)

```
Worker <Thread(worker 2, started 130283824404752)> running with argument 1
Worker <Thread(worker 3, started 130283816012048)> running with argument 2
Worker <Thread(worker 4, started 130283807619344)> running with argument 3
Worker <Thread(worker 5, started 130283799226640)> running with argument 4
Worker <Thread(worker 1, started 130283832797456)> running with argument 5
...
```

Consulte la documentación del módulo para más detalles; la clase `Queue` proporciona una interfaz llena de características.

4.3.4 ¿Qué tipos de mutación de valores globales son *thread-safe*?

Un *global interpreter lock* (GIL) se usa internamente para asegurar que sólo un hilo corre a la vez en la VM de Python. En general, Python ofrece cambiar entre hilos sólo en instrucciones bytecode; la frecuencia con la que cambia se puede fijar vía `sys.setswitchinterval()`. Cada instrucción bytecode y por lo tanto, toda la implementación de código C alcanzada por cada instrucción, es atómica desde el punto de vista de un programa Python.

En teoría, esto significa que un informe exacto requiere de un conocimiento exacto de la implementación en bytecode de la PVM. En la práctica, esto significa que las operaciones entre variables compartidas de tipos de datos *built-in* (enteros, listas, diccionarios, etc.) que «parecen atómicas» realmente lo son.

Por ejemplo, las siguientes operaciones son todas atómicas (L, L1, L2 son listas, D, D1, D2 son diccionarios, *x, y* son objetos, *i, j* son enteros):

```
L.append(x)
L1.extend(L2)
x = L[i]
x = L.pop()
L1[i:j] = L2
L.sort()
x = y
x.field = y
D[x] = y
D1.update(D2)
D.keys()
```

Estas no lo son:

```
i = i+1
L.append(L[-1])
L[i] = L[j]
D[x] = D[x] + 1
```

Las operaciones que reemplazan otros objetos pueden invocar el método `__del__()` de esos otros objetos cuando su número de referencias alcance cero, y eso puede afectar a otras cosas. Esto es especialmente cierto para las actualizaciones en masa de diccionarios y listas. Cuando se esté en duda, ¡use un mutex!

4.3.5 ¿Podemos deshacernos del *Global Interpreter Lock*?

El *global interpreter lock* (GIL) se percibe a menudo como un obstáculo en el despliegue de Python sobre máquinas servidoras finales de múltiples procesadores, porque un programa Python multihilo efectivamente sólo usa una CPU, debido a la exigencia de que (casi) todo el código Python sólo puede correr mientras el GIL esté activado.

En los días de Python 1.5, Greg Stein de hecho implementó un conjunto amplio de parches (los parches «libres de hilo») que eliminaba el GIL y lo reemplazaba con un bloqueo de grano fino. Adam Olsen hizo recientemente un experimento similar con su proyecto python-safethread. Desafortunadamente, ambos experimentos mostraron una aguda caída en el rendimiento (al menos del 30% o más baja), debido a la cantidad de bloqueos de grano fino necesarios para compensar la eliminación del GIL.

¡Esto no significa que no pueda hacer buen uso de Python en máquinas de múltiples CPU! Usted sólo tiene que ser creativo a la hora de dividir el trabajo entre múltiples *procesos* en vez de entre múltiples *hilos*. La clase `ProcessPoolExecutor` del nuevo módulo `concurrent.futures` proporciona una manera sencilla de hacer esto; el módulo `multiprocessing` proporciona una API de bajo nivel en caso de que se quiera tener un mayor control sobre el despacho de las tareas.

El uso sensato de extensiones C también ayudará; si usa una extensión C para ejecutar una tarea que consume mucho tiempo, la extensión puede liberar al GIL mientras el hilo de ejecución esté en el código C y permite a otros hilos hacer trabajo. Algunos módulos de la biblioteca estándar tales como `zlib` y `hashlib` ya lo hacen.

Se ha sugerido que el GIL debería ser un bloqueo por estado de intérprete, en vez de realmente global; luego los intérpretes no serían capaces de compartir objetos. Desafortunadamente, esto tampoco es probable que ocurra. Sería una tremenda cantidad de trabajo, porque muchas implementaciones de objetos actualmente tienen un estado global. Por ejemplo, los enteros pequeños y las cadenas pequeñas están *cacheadas*; estas *caches* se tendrían que mover al estado del intérprete. Otros tipos de objetos tienen su propia lista libre; estas listas libres se tendrían que mover al estado del intérprete. Y así sucesivamente.

Y dudo de si se puede hacer en tiempo finito, porque el mismo problema existe para extensiones de terceros. Es probable que las extensiones de terceros se escriban más rápido de lo que se puedan convertir para almacenar todo su estado global en el estado del intérprete.

Y finalmente, una vez que tenga múltiples intérpretes sin compartir ningún estado, ¿qué habrá ganado sobre correr cada intérprete en un proceso separado?

4.4 Entrada y Salida

4.4.1 ¿Cómo borro un fichero? (Y otras preguntas sobre ficheros...)

Use `os.remove(filename)` o `os.unlink(filename)`; para la documentación, vea el módulo `os`. Las dos funciones son idénticas; `unlink()` es simplemente el nombre de la llamada al sistema UNIX para esta función.

Para borrar un directorio, use `os.rmdir()`; use `os.mkdir()` para crear uno. `os.makedirs(path)` creará cualquier directorio intermedio que no exista en `path`. `os.removedirs(path)` borrará los directorios intermedios siempre y cuando estén vacíos; si quiere borrar un árbol de directorios completo y sus contenidos, use `shutil.rmtree()`.

Para renombrar un fichero, use `os.rename(old_path, new_path)`.

Para truncar un fichero, ábralo usando `f = open(filename, "rb+")`, y use `f.truncate(offset)`; el desplazamiento toma por defecto la posición actual de búsqueda. También existe `os.ftruncate(fd, offset)` para ficheros abiertos con `os.open()`, donde *fd* es el descriptor del fichero (un entero pequeño).

El módulo `shutil` también contiene distintas funciones para trabajar con ficheros incluyendo `copyfile()`, `copytree()` y `rmtree()`.

4.4.2 ¿Cómo copio un fichero?

The `shutil` module contains a `copyfile()` function. Note that on MacOS 9 it doesn't copy the resource fork and Finder info.

4.4.3 ¿Cómo leo (o escribo) datos binarios?

Para leer o escribir formatos binarios de datos complejos, es mejor usar el módulo `struct`. Esto le permite tomar una cadena de texto que contiene datos binarios (normalmente números) y convertirla a objetos de Python; y viceversa.

Por ejemplo, el siguiente código lee de un fichero dos enteros de 2-bytes y uno de 4-bytes en formato *big-endian*:

```
import struct

with open(filename, "rb") as f:
    s = f.read(8)
    x, y, z = struct.unpack(">hhl", s)
```

El “>” en la cadena de formato fuerza los datos a *big-endian*; la letra “h” lee un «entero corto» (2 bytes), y “l” lee un «entero largo» (4 bytes) desde la cadena de texto.

Para datos que son más regulares (por ejemplo una lista homogénea de enteros o flotantes), puede también usar el módulo `array`.

Nota: Para leer y escribir datos binarios, es obligatorio abrir el fichero en modo binario (aquí, pasando `"rb"` a `open()`). Si, en cambio, usa `"r"` (por defecto), el fichero se abrirá en modo texto y `f.read()` retornará objetos `str` en vez de objetos `bytes`.

4.4.4 No consigo usar os.read() en un *pipe* creado con os.popen(); ¿por qué?

`os.read()` es una función de bajo nivel que recibe un descriptor de fichero, un entero pequeño representando el fichero abierto. `os.popen()` crea un objeto fichero de alto nivel, el mismo tipo que retorna la función *built-in* `open()`. Así, para leer *n* bytes de un *pipe p* creado con `os.popen()`, necesita usar `p.read(n)`.

4.4.5 ¿Cómo accedo al puerto serial (RS232)?

Para *Win32*, POSIX (Linux, BSD, etc.), Jython:

http://pyserial.sourceforge.net

Para Unix, vea una publicación Usenet de Mitch Chapman:

https://groups.google.com/groups?selm=34A04430.CF9@ohioee.com

4.4.6 ¿Por qué al cerrar sys.stdout (stdin, stderr) realmente no se cierran?

Los *objetos de tipo fichero* en Python son una capa de abstracción de alto nivel sobre los descriptores de ficheros de bajo nivel de C.

Para la mayoría de objetos de tipo fichero que cree en Python vía la función *built-in* `open()`, `f.close()` marca el objeto de tipo fichero Python como ya cerrado desde el punto de vista de Python, y también ordena el cierre del descriptor de fichero subyacente en C. Esto además ocurre automáticamente en el destructor de `f`, cuando `f` se convierte en basura.

Pero *stdin*, *stdout* y *stderr* se tratan de manera especial en Python, debido a un estatus especial que también tienen en C. Ejecutando `sys.stdout.close()` marca el objeto fichero de nivel Python para ser cerrado, pero *no* cierra el descriptor de fichero asociado en C.

Para cerrar el descriptor de fichero subyacente en C para uno de estos tres casos, debería primero asegurarse de que eso es realmente lo que quiere hacer (por ejemplo, puede confundir módulos de extensión intentado hacer *I/O*). Si es así, use `os.close()`:

```
os.close(stdin.fileno())
os.close(stdout.fileno())
os.close(stderr.fileno())
```

O puede usar las constantes numéricas 0, 1 y 2, respectivamente.

4.5 Programación de Redes/Internet

4.5.1 ¿Qué herramientas de Python existen para WWW?

Vea los capítulos titulados internet y netdata en el manual de referencia de bibliotecas. Python tiene muchos módulos que le ayudarán a construir sistemas web del lado del servidor y del lado del cliente.

Paul Boddie mantiene un resumen de los *frameworks* disponibles en https://wiki.python.org/moin/WebProgramming.

Cameron Laird mantiene un conjunto útil de páginas sobre tecnologías web Python en http://phaseit.net/claird/comp.lang.python/web_python.

4.5.2 ¿Cómo puedo imitar un envío de formulario CGI (*METHOD=POST*)?

Me gustaría recuperar páginas web que son resultado del envío de un formulario. ¿Existe algún código que me permita hacer esto fácilmente?

Sí. Aquí hay un ejemplo sencillo que usa urllib.request:

```
#!/usr/local/bin/python

import urllib.request

# build the query string
qs = "First=Josephine&MI=Q&Last=Public"

# connect and send the server a path
req = urllib.request.urlopen('http://www.some-server.out-there'
                             '/cgi-bin/some-cgi-script', data=qs)
with req:
    msg, hdrs = req.read(), req.info()
```

Nótese que para operaciones POST de tipo *percent-encoded*, las cadenas de consulta tienen que estar entrecomilladas usando `urllib.parse.urlencode()`. Por ejemplo, para enviar `name=Guy Steele, Jr.`:

```
>>> import urllib.parse
>>> urllib.parse.urlencode({'name': 'Guy Steele, Jr.'})
'name=Guy+Steele%2C+Jr.'
```

Ver también:

urllib-howto para ejemplos más detallados.

4.5.3 ¿Qué modulo debería usar para generación de HTML?

Puede encontrar una colección de enlaces útiles en la página wiki de programación web.

4.5.4 ¿Cómo envío correo desde un script Python?

Use el módulo `smtplib` de la biblioteca estándar.

Aquí hay un remitente simple interactivo que lo usa. Este método trabajará en cualquier máquina que soporte un *listener SMTP*.

```
import sys, smtplib

fromaddr = input("From: ")
toaddrs  = input("To: ").split(',')
print("Enter message, end with ^D:")
msg = ''
while True:
    line = sys.stdin.readline()
    if not line:
        break
    msg += line

# The actual mail send
server = smtplib.SMTP('localhost')
server.sendmail(fromaddr, toaddrs, msg)
server.quit()
```

Una alternativa sólo para UNIX es usar *sendmail*. La ubicación del programa *sendmail* varía entre sistemas; algunas veces está en `/usr/lib/sendmail`, otras veces en `/usr/sbin/sendmail`. El manual de *sendmail* le ayudará. Aquí hay un ejemplo de código:

```
import os

SENDMAIL = "/usr/sbin/sendmail"  # sendmail location
p = os.popen("%s -t -i" % SENDMAIL, "w")
p.write("To: receiver@example.com\n")
p.write("Subject: test\n")
p.write("\n")  # blank line separating headers from body
p.write("Some text\n")
p.write("some more text\n")
sts = p.close()
if sts != 0:
    print("Sendmail exit status", sts)
```

4.5.5 ¿Cómo evito el bloqueo en el método *connect()* de un *socket*?

El módulo `select` es mayoritariamente usado para ayudar con entrada/salida de sockets.

Para prevenir el bloqueo en la conexión TCP, puede establecer el socket en modo no bloqueante. Luego cuando ejecute `connect()`, conectará inmediatamente (improbable) u obtendrá una excepción que contiene el número de error como `.errno`. `errno.EINPROGRESS` indica que la conexión está en curso, pero aún no ha terminado. Diferentes sistemas operativos retornarán diferentes valores, así que debe comprobar cuál es el retornado en su sistema.

Puede usar el método `connect_ex()` para evitar crear una excepción. Retornará simplemente el número de error. Para sondear, puede llamar más tarde a `connect_ex()` de nuevo – `0` o `errno.EISCONN` indican que está conectado – o puede pasar este socket a *select* para comprobar si se puede escribir en él.

Nota: El módulo `asyncore` ofrece una enfoque de tipo *framework* al problema de escribir código de red no bloqueante. La biblioteca de terceros Twisted es una alternativa popular y ofrece muchas capacidades.

4.6 Bases de datos

4.6.1 ¿Hay paquetes para interfaces a bases de datos en Python?

Sí.

Interfaces a *hashes* basados en disco tales como `DBM` y `GDBM` están también incluidas en Python estándar. También hay un módulo `sqlite3`, que proporciona una base de datos relacional ligera basada en disco.

Está disponible el soporte para la mayoría de bases de datos relacionales. Vea la página wiki de Programación de Bases de datos para más detalles.

4.6.2 ¿Cómo implementar objetos persistentes en Python?

El módulo de biblioteca `pickle` soluciona esto de una forma muy general (aunque todavía no puede almacenar cosas como ficheros abiertos, sockets o ventanas), y el módulo de biblioteca `shelve` usa *pickle* y *(g)dbm* para crear mapeos persistentes que contienen objetos arbitrarios Python.

4.7 Matemáticas y Numérica

4.7.1 ¿Cómo genero números aleatorios en Python?

El módulo estándar `random` implementa un generador de números aleatorios. El uso es simple:

```
import random
random.random()
```

Esto retorna un número flotante aleatorio en el rango [0, 1).

Hay también muchos otros generadores especializados en este módulo, tales como:

- `randrange(a, b)` selecciona un entero en el rango [a, b).
- `uniform(a, b)` selecciona un número flotante en el rango [a, b).
- `normalvariate(mean, sdev)` muestrea una distribución normal (*Gausiana*).

Algunas funciones de alto nivel operan directamente sobre secuencias, tales como:

- `choice(S)` selecciona un elemento aleatorio de una secuencia dada
- `shuffle(L)` reorganiza una lista in-situ, es decir, la permuta aleatoriamente

También hay una clase `Random` que usted puede instanciar para crear múltiples generadores independientes de valores aleatorios.

CAPÍTULO 5

Extendiendo/Embebiendo FAQ

5.1 ¿Puedo crear mis propias funciones en C?

Si, puedes crear módulos incorporados que contengan funciones, variables, excepciones y incluso nuevos tipos en C. Esto esta explicado en el documento extending-index.

La mayoría de los libros intermedios o avanzados de Python también tratan este tema.

5.2 ¿Puedo crear mis propias funciones en C++?

Si, utilizando las características de compatibilidad encontradas en C++. Coloca `extern "C" { ... }` alrededor los archivos incluidos Python y pon `extern "C"` antes de cada función que será llamada por el interprete Python. Objetos globales o estáticos C++ con constructores no son una buena idea seguramente.

5.3 Escribir en C es difícil; ¿no hay otra alternativa?

Hay un número de alternativas a escribir tus propias extensiones C, dependiendo en que estés tratando de hacer.

Cython y su relativo Pyrex son compiladores que aceptan una forma de Python ligeramente modificada y generan el código C correspondiente. Cython y *Pyrex* hacen posible escribir una extensión sin tener que aprender la API de Python C.

Si necesitas hacer una interfaz a alguna biblioteca C o C++ que no posee aún extensión Python, puedes intentar empaquetar los tipo de datos de la biblioteca con una herramienta como SWIG. SIP, CXX Boost, o Weave también son alternativas para empaquetar bibliotecas C++.

5.4 ¿Cómo puedo ejecutar declaraciones arbitrarias de Python desde C?

La función de más alto nivel para hacer esto es `PyRun_SimpleString()` que toma un solo argumento de cadena de caracteres para ser ejecutado en el contexto del módulo `__main__` y retorna `0` si tiene éxito y `-1` cuando ocurre una excepción (incluyendo `SyntaxError`). Si quieres mas control, usa `PyRun_String()`; mira la fuente para `PyRun_SimpleString()` en `Python/pythonrun.c`.

5.5 ¿Cómo puedo evaluar una expresión arbitraria de Python desde C?

Call the function `PyRun_String()` from the previous question with the start symbol `Py_eval_input`; it parses an expression, evaluates it and returns its value.

5.6 ¿Cómo extraigo valores C de un objeto Python?

Eso depende del tipo de objeto. Si es una tupla, `PyTuple_Size()` retorna su tamaño, y `PyTuple_GetItem()` retorna el ítem del índice especificado. Las listas tienen funciones similares, `PyListSize()` and `PyList_GetItem()`.

Para bytes `PyBytes_Size()` retorna su tamaño, y `PyBytes_AsStringAndSize()` proporciona un puntero a su valor y tamaño. Nota que los objetos byte de Python pueden contener bytes *null* por lo que la función de C `strlen()` no debe ser utilizada.

Para testear el tipo de un objeto, primero debes estar seguro que no es `NULL`, y luego usa `PyBytes_Check()`, `PyTuple_Check()`, `PyList_Check()`, etc.

También hay una API de alto nivel para objetos Python que son provistos por la supuestamente llamada interfaz "abstracta" – lee `Include/abstract.h` para mas detalles. Permite realizar una interfaz con cualquier tipo de secuencia Python usando llamadas como `PySequence_Length()`, `PySequence_GetItem()`, etc. así como otros protocolos útiles como números (`PyNumber_Index()` et al.) y mapeos en las *PyMapping APIs*.

5.7 ¿Cómo utilizo Py_BuildValue() para crear una tupla de un tamaño arbitrario?

No puedes hacerlo. Utiliza a cambio `PyTuple_Pack()`.

5.8 ¿Cómo puedo llamar un método de un objeto desde C?

Se puede utilizar la función `PyObject_CallMethod()` para llamar a un método arbitrario de un objeto. Los parámetros son el objeto, el nombre del método a llamar, una cadena de caracteres de formato como las usadas con `Py_BuildValue()`, y los valores de argumento

```
PyObject *
PyObject_CallMethod(PyObject *object, const char *method_name,
                    const char *arg_format, ...);
```

Esto funciona para cualquier objeto que tenga métodos – sean estos incorporados o definidos por el usuario. Eres responsable si eventualmente usas `Py_DECREF()` en el valor de retorno.»

Para llamar, por ejemplo, un método «seek» de un objeto archivo con argumentos 10, 0 (considerando que puntero del objeto archivo es «f»):

```
res = PyObject_CallMethod(f, "seek", "(ii)", 10, 0);
if (res == NULL) {
        ... an exception occurred ...
}
else {
        Py_DECREF(res);
}
```

Note que debido a `PyObject_CallObject()` *siempre* necesita una tupla para la lista de argumento, para llamar una función sin argumentos, deberás pasar «()» para el formato, y para llamar a una función con un solo argumento, encierra el argumento entre paréntesis, por ejemplo «(i)».

5.9 ¿Cómo obtengo la salida de PyErr_Print() (o cualquier cosa que se imprime a stdout/stderr)?

En código Python, define un objeto que soporta el método `write()`. Asigna este objeto a `sys.stdout` y `sys.stderr`. Llama a print_error, o solo permite que el mecanismo estándar de rastreo funcione. Luego, la salida se hará cuando invoques `write()`.

La manera mas fácil de hacer esto es usar la clase `io.StringIO`.

```
>>> import io, sys
>>> sys.stdout = io.StringIO()
>>> print('foo')
>>> print('hello world!')
>>> sys.stderr.write(sys.stdout.getvalue())
foo
hello world!
```

Un objeto personalizado para hacer lo mismo se vería así:

```
>>> import io, sys
>>> class StdoutCatcher(io.TextIOBase):
...     def __init__(self):
...         self.data = []
...     def write(self, stuff):
...         self.data.append(stuff)
...
>>> import sys
>>> sys.stdout = StdoutCatcher()
>>> print('foo')
>>> print('hello world!')
>>> sys.stderr.write(''.join(sys.stdout.data))
foo
hello world!
```

5.10 ¿Cómo accedo al módulo escrito en Python desde C?

Puedes obtener un puntero al módulo objeto de esta manera:

```
module = PyImport_ImportModule("<modulename>");
```

Si el módulo todavía no se importó, (por ejemplo aún no esta presente en `sys.modules`), esto inicializa el módulo, de otra forma simplemente retorna el valor de `sys.modules["<modulename>"]`. Nota que no entra el módulo a ningún espacio de nombres (*namespace*) –solo asegura que fue inicializado y guardado en `sys.modules`.

Puedes acceder luego a los atributos del módulo (por ejemplo a cualquier nombre definido en el módulo) de esta forma:

```
attr = PyObject_GetAttrString(module, "<attrname>");
```

También funciona llamar a `PyObject_SetAttrString()` para asignar a variables en el módulo.

5.11 ¿Cómo hago una interface a objetos C++ desde Python?

Dependiendo de lo que necesites, hay varias maneras de hacerlo. Para hacerlo manualmente empieza por leer the «Extending and Embedding» document.Fíjate que para el sistema de tiempo de ejecución Python, no hay una gran diferencia entre C y C++, por lo que la estrategia de construir un nuevo tipo Python alrededor de una estructura de C de tipo puntero, también funcionará para objetos C++.

Para bibliotecas C++, mira *Escribir en C es difícil; ¿no hay otra alternativa?*.

5.12 He agregado un módulo usando el archivo de configuración y el *make* falla. ¿Porque?

La configuración debe terminar en una nueva linea, si esta no está, entonces el proceso *build* falla. (reparar esto requiere algún truco de linea de comandos que puede ser no muy prolijo, y seguramente el error es tan pequeño que no valdrá el esfuerzo.)

5.13 ¿Cómo puedo depurar una extención?

When using GDB with dynamically loaded extensions, you can't set a breakpoint in your extension until your extension is loaded.

En tu archivo `.gdbinit` (o interactivamente), agrega el comando:

```
br _PyImport_LoadDynamicModule
```

Luego, cuando corras GDB:

```
$ gdb /local/bin/python
gdb) run myscript.py
gdb) continue # repeat until your extension is loaded
gdb) finish   # so that your extension is loaded
gdb) br myfunction.c:50
gdb) continue
```

5.14 Quiero compilar un módulo Python en mi sistema Linux, pero me faltan algunos archivos . ¿Por qué?

La mayoría de las versiones empaquetadas de Python no incluyen el directorio `/usr/lib/python2.x/config/`, que contiene varios archivos que son necesarios para compilar las extensiones Python.

Para *Red Hat*, instala el *python-devel RPM* para obtener los archivos necesarios.

Para Debian, corre `apt-get install python-dev`.

5.15 How do I tell «incomplete input» from «invalid input»?

A veces quieres emular el comportamiento del interprete interactivo de Python, que te da una continuación del *prompt* cuando la entrada esta incompleta (por ejemplo si comenzaste tu instrucción «if» o no cerraste un paréntesis o triples comillas), pero te da un mensaje de error de sintaxis inmediatamente cuando la entrada es invalida.

In Python you can use the `codeop` module, which approximates the parser's behavior sufficiently. IDLE uses this, for example.

La manera mas fácil de hacerlo en C es llamar a `PyRun_InteractiveLoop()` (quizá en un hilo separado) y dejar que el interprete Python gestione la entrada por ti. Puedes también configurar `PyOS_ReadlineFunctionPointer()` para apuntar a tu función de entrada personalizada.

De todas maneras a veces debes correr el interprete embebido de Python en el mismo hilo que tu *api rest*, y puedes permitir que `PyRun_InteractiveLoop()` termine mientras espera por la entrada del usuario. La primera solución es llamar a `PyParser_ParseString()` y probar con `e.error` igual a `E_EOF`, que significa que la entrada esta incompleta. Aquí hay un fragmento de código ejemplo, que no esta probado, inspirado en el código de Alex Farber:

```
#define PY_SSIZE_T_CLEAN
#include <Python.h>
#include <node.h>
#include <errcode.h>
#include <grammar.h>
#include <parsetok.h>
#include <compile.h>

int testcomplete(char *code)
  /* code should end in \n */
  /* return -1 for error, 0 for incomplete, 1 for complete */
{
  node *n;
  perrdetail e;

  n = PyParser_ParseString(code, &_PyParser_Grammar,
                           Py_file_input, &e);
  if (n == NULL) {
    if (e.error == E_EOF)
      return 0;
    return -1;
  }

  PyNode_Free(n);
  return 1;
}
```

Otra solución es intentar compilar la cadena de caracteres recibida con `Py_CompileString()`. Si compila sin errores, intenta ejecutar el objeto código retornado llamando a `PyEval_EvalCode()`. De otra manera salva el ingreso para después. Si la compilación falla, encuentra si hay algún o si necesita algún ingreso adicional - extrayendo la cadena de caracteres mensaje de la tupla excepción y comparándola con la cadena de caracteres «unexpected EOF while parsing». Aquí hay un ejemplo completo usando la biblioteca *GNU readline* (Seguramente querrás ignorar **SIGINT** mientras llamas a readline()):

```
#include <stdio.h>
#include <readline.h>

#define PY_SSIZE_T_CLEAN
#include <Python.h>
#include <object.h>
#include <compile.h>
#include <eval.h>
```

(continué en la próxima página)

(proviene de la página anterior)

```
int main (int argc, char* argv[])
{
  int i, j, done = 0;                           /* lengths of line, code */
  char ps1[] = ">>> ";
  char ps2[] = "... ";
  char *prompt = ps1;
  char *msg, *line, *code = NULL;
  PyObject *src, *glb, *loc;
  PyObject *exc, *val, *trb, *obj, *dum;

  Py_Initialize ();
  loc = PyDict_New ();
  glb = PyDict_New ();
  PyDict_SetItemString (glb, "__builtins__", PyEval_GetBuiltins ());

  while (!done)
  {
    line = readline (prompt);

    if (NULL == line)                           /* Ctrl-D pressed */
    {
      done = 1;
    }
    else
    {
      i = strlen (line);

      if (i > 0)
        add_history (line);                     /* save non-empty lines */

      if (NULL == code)                         /* nothing in code yet */
        j = 0;
      else
        j = strlen (code);

      code = realloc (code, i + j + 2);
      if (NULL == code)                         /* out of memory */
        exit (1);

      if (0 == j)                               /* code was empty, so */
        code[0] = '\0';                         /* keep strncat happy */

      strncat (code, line, i);                  /* append line to code */
      code[i + j] = '\n';                       /* append '\n' to code */
      code[i + j + 1] = '\0';

      src = Py_CompileString (code, "<stdin>", Py_single_input);

      if (NULL != src)                          /* compiled just fine - */
      {
        if (ps1  == prompt ||                   /* ">>> " or */
            '\n' == code[i + j - 1])            /* "... " and double '\n' */
        {                                               /* so execute it */
          dum = PyEval_EvalCode (src, glb, loc);
          Py_XDECREF (dum);
          Py_XDECREF (src);
          free (code);
          code = NULL;
          if (PyErr_Occurred ())
            PyErr_Print ();
```

(continué en la próxima página)

(proviene de la página anterior)

```
          prompt = ps1;
        }
      }                                         /* syntax error or E_EOF? */
      else if (PyErr_ExceptionMatches (PyExc_SyntaxError))
      {
        PyErr_Fetch (&exc, &val, &trb);         /* clears exception! */

        if (PyArg_ParseTuple (val, "sO", &msg, &obj) &&
            !strcmp (msg, "unexpected EOF while parsing")) /* E_EOF */
        {
          Py_XDECREF (exc);
          Py_XDECREF (val);
          Py_XDECREF (trb);
          prompt = ps2;
        }
        else                                    /* some other syntax error */
        {
          PyErr_Restore (exc, val, trb);
          PyErr_Print ();
          free (code);
          code = NULL;
          prompt = ps1;
        }
      }
      else                                      /* some non-syntax error */
      {
        PyErr_Print ();
        free (code);
        code = NULL;
        prompt = ps1;
      }

      free (line);
    }
  }

  Py_XDECREF(glb);
  Py_XDECREF(loc);
  Py_Finalize();
  exit(0);
}
```

5.16 How do I find undefined g++ symbols __builtin_new or __pure_virtual?

Para cargar dinámicamente módulos de extensión g++, debes recompilar Python, hacer un nuevo *link* usando g++ (cambia LINKCC en el Python Modules Makefile) y enlaza *link* tu extensión usando g++ (por ejemplo *g++ -shared -o mymodule.so mymodule.o*‘).

5.17 ¿Puedo crear una clase objeto con algunos métodos implementado en C y otros en Python (por ejemplo a través de la herencia)?

Si, puedes heredar de clases integradas como `int`, `list`, `dict`, etc.

La biblioteca *Boost Pyhton* (BPL, http://www.boost.org/libs/python/doc/index.html) provee una manera de realizar esto desde C++ (por ejemplo puedes heredar de una clase extensión escrita en C++ usando el BPL).

CAPÍTULO 6

Preguntas frecuentes sobre Python en Windows

6.1 ¿Cómo ejecutar un programa Python en Windows?

No es necesariamente una pregunta simple. Si ya está familiarizado con el lanzamiento de programas desde la línea de comandos de Windows, todo parecerá obvio; de lo contrario, es posible que necesite un poco más de orientación.

A menos que esté utilizando algún tipo de entorno de desarrollo, terminará escribiendo comandos de Windows en lo que se denomina «DOS» o «símbolo del sistema de Windows». En general, puede abrir dicha ventana desde su barra de búsqueda buscando *cmd*. Debería poder reconocer cuándo inició dicha ventana porque verá un símbolo del sistema de Windows, que en general se ve así:

```
C:\>
```

La letra puede ser diferente y puede haber otras cosas seguidas, por lo que también puede verse así:

```
D:\YourName\Projects\Python>
```

dependiendo de la configuración de su computadora y de lo que haya hecho recientemente con ella. Una vez que haya abierto esta ventana, está en camino de iniciar los programas de Python.

Tenga en cuenta que sus scripts de Python deben ser procesados por otro programa llamado «intérprete» de Python. El intérprete lee su script, lo compila en *bytecode* y ejecuta el *bytecode* para ejecutar su programa. Entonces, ¿cómo le das tu código Python al intérprete?

Primero, debe asegurarse de que la ventana del símbolo del sistema reconoce la palabra «python» como una instrucción para iniciar el intérprete. Si abrió un símbolo del sistema, escriba el comando `py` y luego presione la tecla Enter:

```
C:\Users\YourName> py
```

Debería ver algo como esto:

```
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:04:45) [MSC v.1900 32 bit (Intel)]␣
↪on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Has comenzado el intérprete en su «modo interactivo». Esto significa que puede ingresar declaraciones o expresiones de Python de forma interactiva para ejecutarlas. Esta es una de las características más poderosas de Python. Puede

verificar esto ingresando algunos comandos de su elección y ver los resultado:

```
>>> print("Hello")
Hello
>>> "Hello" * 3
'HelloHelloHello'
```

Muchas personas usan el modo interactivo como una calculadora práctica pero altamente programable. Cuando desee finalizar su sesión interactiva de Python, llame a la función `exit()` o mantenga presionada la tecla `Ctrl` mientras ingresa una `Z`, luego presione la tecla «`Enter`» para regresar a su símbolo del sistema de Windows.

Es posible que haya notado una nueva entrada en su menú Inicio, como *Inicio ▸ Programas ▸ Python 3.x ▸ Python (línea de comando)* que hace que vea el mensaje `>>>` en una nueva ventana. Si es así, la ventana desaparecerá cuando llame a la función `exit()` o presione la combinación `Ctrl-Z`; Windows ejecuta un comando «python» en la ventana y lo cierra cuando cierra el intérprete.

Ahora que sabemos que se reconoce el comando `py`, puede darle su script Python. Debe proporcionar la ruta absoluta o relativa de la secuencia de comandos de Python. Digamos que su script Python se encuentra en su escritorio y se llama `hello.py`, y su símbolo del sistema está abierto en su directorio de inicio, por lo que verá algo como:

```
C:\Users\YourName>
```

Entonces, le pedirá al comando `py` que le dé su script a Python escribiendo `py` seguido de la ruta de su script:

```
C:\Users\YourName> py Desktop\hello.py
hello
```

6.2 ¿Cómo hacer que los scripts de Python sean ejecutables?

En Windows, el instalador de Python asocia la extensión *.py* con un tipo de archivo *(Python.File)* y un comando que inicia el intérprete (`D:\Archivos de programa\Python\python.exe "%1" %*`) . Esto es suficiente para poder ejecutar scripts de Python desde la línea de comandos ingresando “foo.py”. Si desea poder ejecutar los scripts simplemente escribiendo “foo” sin la extensión, debe agregar .py a la variable del entorno PATHEXT.

6.3 ¿Por qué Python tarda en comenzar?

Normalmente en Windows, Python se inicia muy rápidamente, pero a veces los informes de error indican que Python de repente comienza a tardar mucho tiempo en iniciarse. Es aún más desconcertante porque Python funcionará correctamente con otros Windows configurados de manera idéntica.

El problema puede provenir de un antivirus mal configurado. Se sabe que algunos antivirus duplican el tiempo de arranque cuando se configuran para verificar todas las lecturas del sistema de archivos. Intente verificar si los antivirus de las dos máquinas están configurados correctamente de manera idéntica. *McAfee* es especialmente problemático cuando se configura para verificar todas las lecturas del sistema de archivos.

6.4 ¿Cómo hacer un ejecutable a partir de un script de Python?

See cx_Freeze and py2exe, both are distutils extensions that allow you to create console and GUI executables from Python code.

6.5 ¿Es un archivo *.pyd lo mismo que una DLL?

Sí, los archivos *.pyd* son archivos dll, pero hay algunas diferencias. Si tiene una DLL llamada `foo.pyd`, debe tener una función `PyInit_foo()`. Luego puede escribir en Python «import foo» y Python buscará el archivo *foo.pyd* (así como *foo.py* y *foo.pyc*); si lo encuentra, intentará llamar a `PyInit_foo()` para inicializarlo. No vincules tu .exe con *foo.lib* porque en este caso Windows necesitará la DLL.

Tenga en cuenta que la ruta de búsqueda para *foo.pyd* es PYTHONPATH, es diferente de la que usa Windows para buscar *foo.dll*. Además, *foo.pyd* no necesita estar presente para que su programa se ejecute, mientras que si ha vinculado su programa con una dll, esto es necesario. Por supuesto, *foo.pyd* es necesario si escribes `import foo`. En una DLL, el enlace se declara en el código fuente con `__declspec(dllexport)`. En un *.pyd*, la conexión se define en una lista de funciones disponibles.

6.6 ¿Cómo puedo integrar Python en una aplicación de Windows?

La integración del intérprete de Python en una aplicación de Windows se puede resumir de la siguiente manera:

1. _No_ compile Python directamente en su archivo .exe. En Windows, Python debe ser una DLL para poder importar módulos que son DLL en sí mismos (esto constituye información esencial no documentada). En su lugar, haga un enlace al archivo `python`*NN*`.dll`; que generalmente se encuentra en `C:\Windows\System`. *NN* es la versión de Python, por ejemplo «33» para Python 3.3.

   Puede vincular a Python de dos maneras diferentes. Un enlace en tiempo de carga significa apuntar al archivo `python`*NN*`.lib`, mientras que un enlace en tiempo de ejecución significa apuntar al archivo `python`*NN*`.dll`. (Nota general: el archivo `python`*NN*`.lib` es la llamada «lib de importación» correspondiente para el archivo `python`*NN*`.dll`. Simplemente define enlaces simbólicos para el enlazador).

   El enlace en tiempo real simplifica enormemente las opciones de enlace; Todo sucede en el momento de la ejecución. Su código debe cargar el archivo `python`*NN*`.dll` utilizando la rutina de Windows `LoadLibraryEx()`. El código también debe usar rutinas de acceso y datos en el archivo `python`*NN*`.dll` (es decir, las API C de Python) usando punteros obtenidos por la rutina de Windows `GetProcAddress()`. Las macros pueden hacer que el uso de estos punteros sea transparente para cualquier código C que llame a rutinas en la API C de Python.

   Nota *Borland*: convierta el archivo `python`*NN*`.lib` al formato OMF usando Coff2Omf.exe primero.

2. Si está utilizando *SWIG*, es fácil crear un «complemento» de Python que hará que los datos y métodos de la aplicación estén disponibles para Python. *SWIG* se encargará de todos los detalles aburridos para usted. El resultado es un código C que se vincula *adentro* de su archivo.exe (!) _No_ necesita crear un archivo DLL, y también simplifica la vinculación.

3. *SWIG* creará una función de inicialización (función en C) cuyo nombre depende del nombre del complemento. Por ejemplo, si el nombre del módulo es leo, la función *init* se llamará *initleo()*. Si está utilizando clases *shadow SWIG*, como debería, la función *init* se llamará *initleoc()*. Esto inicializa una clase auxiliar invisible utilizada por la clase *shadow*.

   ¡La razón por la que puede vincular el código C en el paso 2 en su archivo .exe es que llamar a la función de inicialización es equivalente a importar el módulo a Python! (Este es el segundo hecho clave indocumentado).

4. En resumen, puede usar el siguiente código para inicializar el intérprete de Python con su complemento.

```
#include "python.h"
...
Py_Initialize();  // Initialize Python.
initmyAppc();  // Initialize (import) the helper class.
PyRun_SimpleString("import myApp");  // Import the shadow class.
```

5. Hay dos problemas con la API de Python C que aparecerán si utiliza un compilador que no sea *MSVC*, el compilador utilizado para construir *pythonNN.dll*.

Problema 1: Las llamadas funciones de «Muy Alto Nivel» que toman los argumentos FILE * no funcionarán en un entorno de compilación múltiple porque cada compilador tendrá una noción diferente de la estructura FILE. Desde el punto de vista de la implementación, estas son funciones de muy _bajo_ nivel.

Problema 2: *SWIG* genera el siguiente código al generar contenedores para cancelar las funciones:

```
Py_INCREF(Py_None);
_resultobj = Py_None;
return _resultobj;
```

Por desgracia, *Py_None* es una macro que se desarrolla con referencia a una estructura de datos compleja llamada *_Py_NoneStruct* dentro de *pythonNN.dll*. Nuevamente, este código fallará en un entorno de múltiples compiladores. Reemplace este código con:

```
return Py_BuildValue("");
```

Es posible utilizar el comando *SWIG* `%typemap` para realizar el cambio automáticamente, aunque no he logrado que funcione (soy un principiante con *SWIG*).

6. Usar una secuencia de comandos de shell Python para crear una ventana de intérprete de Python desde su aplicación de Windows no es una buena idea; la ventana resultante será independiente del sistema de ventanas de su aplicación. Usted (o la clase *wxPythonWindow*) debería crear una ventana de intérprete «nativa». Es fácil conectar esta ventana al intérprete de Python. Puede redirigir la entrada/salida de Python a cualquier objeto que admita lectura y escritura, por lo que todo lo que necesita es un objeto de Python (definido en su complemento) que contenga los métodos de *read()* y *write()*.

6.7 ¿Cómo puedo evitar que mi editor inserte pestañas en mi archivo fuente de Python?

Las preguntas frecuentes no recomiendan el uso de pestañas y la guía de estilo de Python, **PEP 8**, recomienda el uso de 4 espacios para distribuir el código de Python. Este es también el modo predeterminado de Emacs con Python.

Sea cual sea su editor, mezclar pestañas y espacios es una mala idea. *MSVC* no es diferente en este aspecto, y se puede configurar fácilmente para usar espacios: vaya a *Tools* ‣ *Options* ‣ *Tabs*, y para el tipo de archivo «Default», debe establecer «Tab size» and «Indent size» en 4, luego seleccione Insertar espacios.

Python lanzará los errores `IndentationError` o `TabError` si una combinación de tabulación y sangría es problemática al comienzo de la línea. También puede usar el módulo `tabnanny` para detectar estos errores.

6.8 ¿Cómo verifico una pulsación de tecla sin bloquearla?

Use the `msvcrt` module. This is a standard Windows-specific extension module. It defines a function `kbhit()` which checks whether a keyboard hit is present, and `getch()` which gets one character without echoing it.

CAPÍTULO 7

Preguntas frecuentes sobre la Interfaz Gráfica de Usuario (*GUI*)

7.1 Preguntas generales de la GUI

7.2 ¿Qué conjuntos de herramientas de GUI independientes por plataforma existen para Python?

Dependiendo de la plataforma(s) que esté apuntando, hay varias. Algunas de ellas aún no han sido llevadas a Python 3. Al menos *Tkinter* y *Qt* son conocidas por ser compatibles con Python 3.

7.2.1 Tkinter

Los empaquetados estándar de Python incluyen una interfaz orientada a objetos para el conjunto de widgets de Tcl/Tk, llamada tkinter. Esta es probablemente la más fácil de instalar (ya que viene incluida con la mayoría de distribuciones binarias de Python) y usar. Para obtener más información sobre Tk, incluyendo referencias a la fuente, ver la Tcl/Tk página de inicio. Tcl/Tk es totalmente potable a Mac OSX, Windows y plataformas Unix.

7.2.2 wxWidgets

wxWidgets (https://www.wxwidgets.org) es una biblioteca de clase GUI portátil y gratuita escrita en C ++ que proporciona una apariencia nativa en varias plataformas, con Windows, Mac OS X, GTK, X11, todas listadas como objetivos estables actuales. Los enlaces a lenguajes están disponibles para varios lenguajes, incluidos Python, Perl, Ruby, etc.

wxPython es el enlace de Python para wxwidgets. Si bien a menudo va un poco por detrás de las versiones oficiales de wxWidgets, también ofrece una serie de características a través de extensiones puras de Python que no están disponibles en otros enlaces de lenguajes. Existe una comunidad activa de usuarios y desarrolladores de wxPython.

Tanto wxWidgets como wxPython son software gratuito, de código abierto, con licencias permisivas que permiten su uso en productos comerciales, así como en *freeware* o *shareware*.

7.2.3 Qt

Hay enlaces disponibles para el conjunto de herramientas Qt (usando PyQt o PySide) y para KDE(PyKDE4). PyQt es actualmente más maduro que PySide, pero debe comprar una licencia PyQt de Riverbank Computing si desea escribir aplicaciones propietarias. PySide es gratuito para todas las aplicaciones.

Qt 4.5 en adelante tiene licencia bajo la licencia LGPL; además, las licencias comerciales están disponibles desde The Qt Company.

7.2.4 Gtk+

Los GObject introspection bindings para Python le permiten escribir aplicaciones GTK + 3 .También hay un Tutorial Python GTK+ 3.

Los enlaces más antiguos de PyGtk para el conjunto de herramientas Gtk+ 2 han sido implementado por James Henstridge; ver <http://www.pygtk.org>.

7.2.5 Kivy

Kivy es una biblioteca GUI multiplataforma que admite sistemas operativos de escritorio (Windows, macOS, Linux) y dispositivos móviles (Android, iOS). Está escrito en Python y Cython, y puede usar una gama de *backends* de ventanas.

Kivy es un software gratuito y de código abierto distribuido bajo la licencia MIT.

7.2.6 FLTK

Los enlaces de Python para el conjunto de herramientas de FLTK, un sistema de ventanas multiplataforma simple pero potente y maduro, están disponibles en el proyecto PyFLTK.

7.2.7 OpenGL

Para abrir enlaces de OpenGL, ver PyOpenGL.

7.3 ¿Qué conjuntos de herramientas de GUI específicas por plataforma existen para Python?

Al instalar PyObjc Objective-C bridge, los programas de Python pueden usar librerías de Mac OS X's Cocoa.

Pythonwin por Mark Hammond incluye una interfaz para las Clases de Microsoft Foundation y un entorno de programación Python que está escrito principalmente en Python usando las clases MFC.

7.4 Preguntas de Tkinter

7.4.1 ¿Cómo congelo las aplicaciones de Tkinter?

Freeze es una herramienta para crear aplicaciones independientes. Al congelar aplicaciones Tkinter, las aplicaciones no serán realmente independientes, ya que la aplicación seguirá necesitando las bibliotecas Tcl y Tk.

Una solución es enviar la aplicación con las bibliotecas Tcl y Tk, y apuntarlas en tiempo de ejecución utilizando `TCL_LIBRARY` y las variables de entorno `TK_LIBRARY`.

Para obtener aplicaciones verdaderamente independientes, los *scripts* Tcl que forman la biblioteca también deben integrarse en la aplicación. Una herramienta compatible es SAM (módulos independientes), que forma parte de la distribución Tix (http://tix.sourceforge.net/).

Construya Tix con SAM habilitado, realice la llamada apropiada a `Tclsam_init()`, etc. dentro de Python `Modules/tkappinit.c`, y enlace con libtclsam libtclsam y libtksam (también puede incluir las bibliotecas Tix).

7.4.2 ¿Puedo tener eventos Tk manejados mientras espero por *I/O*?

En plataformas que no sean Windows, sí, ¡y ni siquiera necesita hilos! Pero tendrá que reestructurar un poco su código de I/O. Tk tiene el equivalente de la llamada Xt `XtAddInput()`, que le permite registrar una función de *callback* que se llamará desde el bucle principal de Tk cuando sea posible I/O en un descriptor de archivo. Ver tkinter-file-handlers.

7.4.3 No puedo hacer que los atajos de teclado funcionen en Tkinter: ¿por qué?

Una queja que se escucha con frecuencia es que los controladores de eventos vinculados a eventos con el método `bind()` no se manejan incluso cuando se presiona la tecla adecuada.

La causa más común es que el widget al que se aplica el atajo no tiene enfoque de teclado. Consulte la documentación de Tk para el comando de *focus*. Por lo general, un *widget* recibe el foco del teclado haciendo clic en él (pero no para las etiquetas; consulte la opción *takefocus*).

CAPÍTULO 8

«¿Por qué está Python instalado en mi ordenador?» FAQ

8.1 ¿Qué es Python?

Python es un lenguaje de programación. Se usa para muchas aplicaciones diferentes. Se utiliza en algunas escuelas secundarias y universidades como un lenguaje de programación introductorio porque Python es fácil de aprender, pero también es utilizado por desarrolladores de software profesionales en lugares como Google, NASA, y Lucasfilm Ltd.

Si desea aprender más sobre Python, comience con la Guía del principiante de Python.

8.2 ¿Por qué Python está instalado en mi máquina?

Si encuentras Python instalado en tu sistema pero no recuerdas haberlo instalado, hay varias maneras posibles en las que podría haber llegado ahí.

- Tal vez otro usuario de la computadora quiso aprender a programar y la instaló.
- Una aplicación de terceros instalada en la máquina podría haber sido escrita en Python e incluir una instalación de Python. Hay muchas aplicaciones de este tipo, desde programas GUI hasta servidores de red y scripts administrativos.
- Algunas máquinas Windows también tienen Python instalado. Al momento de escribir este artículo, sabemos que las computadoras de Hewlett-Packard y Compaq incluyen Python. Aparentemente algunas de las herramientas administrativas de HP/Compaq están escritas en Python.
- Muchos sistemas operativos compatibles con Unix, como Mac OS X y algunas distribuciones de Linux, tienen Python instalado por defecto; está incluido en la instalación base.

8.3 ¿Puedo eliminar Python?

Eso depende de dónde vino Python.

Si alguien lo instaló deliberadamente, puede quitarlo sin dañar nada. En Windows, utilice el icono Agregar o quitar programas en el Panel de control.

Si Python fue instalado por una aplicación de terceros, también puede eliminarlo, pero esa aplicación ya no funcionará. Deberías usar el desinstalador de esa aplicación en lugar de eliminar Python directamente.

Si Python vino con su sistema operativo, no se recomienda quitarlo. Si lo eliminas, las herramientas escritas en Python ya no funcionarán, y algunas de ellas pueden ser importantes para ti. Reinstalar todo el sistema sería entonces necesario para arreglar las cosas de nuevo.

APÉNDICE A

Glosario

>>> El prompt en el shell interactivo de Python por omisión. Frecuentemente vistos en ejemplos de código que pueden ser ejecutados interactivamente en el intérprete.

. . . Puede referirse a:

- El prompt en el shell interactivo de Python por omisión cuando se ingresa código para un bloque indentado de código, y cuando se encuentra entre dos delimitadores que emparejan (paréntesis, corchetes, llaves o comillas triples), o después de especificar un decorador.
- La constante incorporada `Ellipsis`.

2to3 Una herramienta que intenta convertir código de Python 2.x a Python 3.x arreglando la mayoría de las incompatibilidades que pueden ser detectadas analizando el código y recorriendo el árbol de análisis sintáctico.

2to3 está disponible en la biblioteca estándar como `lib2to3`; un punto de entrada independiente es provisto como `Tools/scripts/2to3`. Vea 2to3-reference.

clase base abstracta Las clases base abstractas (ABC, por sus siglas en inglés *Abstract Base Class*) complementan al *duck-typing* brindando un forma de definir interfaces con técnicas como `hasattr()` que serían confusas o sutilmente erróneas (por ejemplo con magic methods). Las ABC introduce subclases virtuales, las cuales son clases que no heredan desde una clase pero aún así son reconocidas por `isinstance()` y `issubclass()`; vea la documentación del módulo `abc`. Python viene con muchas ABC incorporadas para las estructuras de datos(en el módulo `collections.abc`), números (en el módulo `numbers`) , flujos de datos (en el módulo `io`) , buscadores y cargadores de importaciones (en el módulo `importlib.abc`) . Puede crear sus propios ABCs con el módulo `abc`.

anotación Una etiqueta asociada a una variable, atributo de clase, parámetro de función o valor de retorno, usado por convención como un *type hint*.

Las anotaciones de variables no pueden ser accedidas en tiempo de ejecución, pero las anotaciones de variables globales, atributos de clase, y funciones son almacenadas en el atributo especial `__annotations__` de módulos, clases y funciones, respectivamente.

Vea *variable annotation*, *function annotation*, **PEP 484** y **PEP 526**, los cuales describen esta funcionalidad.

argumento Un valor pasado a una *function* (o *method*) cuando se llama a la función. Hay dos clases de argumentos:

- *argumento nombrado*: es un argumento precedido por un identificador (por ejemplo, `nombre=`) en una llamada a una función o pasado como valor en un diccionario precedido por `**`. Por ejemplo `3` y `5` son argumentos nombrados en las llamadas a `complex()`:

```
complex(real=3, imag=5)
complex(**{'real': 3, 'imag': 5})
```

- *argumento posicional* son aquellos que no son nombrados. Los argumentos posicionales deben aparecer al principio de una lista de argumentos o ser pasados como elementos de un *iterable* precedido por `*`. Por ejemplo, `3` y `5` son argumentos posicionales en las siguientes llamadas:

```
complex(3, 5)
complex(*(3, 5))
```

Los argumentos son asignados a las variables locales en el cuerpo de la función. Vea en la sección calls las reglas que rigen estas asignaciones. Sintácticamente, cualquier expresión puede ser usada para representar un argumento; el valor evaluado es asignado a la variable local.

Vea también el *parameter* en el glosario, la pregunta frecuente *la diferencia entre argumentos y parámetros*, y **PEP 362**.

administrador asincrónico de contexto Un objeto que controla el entorno visible en un sentencia `async with` al definir los métodos `__aenter__()` `__aexit__()`. Introducido por **PEP 492**.

generador asincrónico Una función que retorna un *asynchronous generator iterator*. Es similar a una función corrutina definida con `async def` excepto que contiene expresiones `yield` para producir series de variables usadas en un ciclo `async for`.

Usualmente se refiere a una función generadora asincrónica, pero puede referirse a un *iterador generador asincrónico* en ciertos contextos. En aquellos casos en los que el significado no está claro, usar los términos completos evita la ambigüedad.

Una función generadora asincrónica puede contener expresiones `await` así como sentencias `async for`, y `async with`.

iterador generador asincrónico Un objeto creado por una función *asynchronous generator*.

Este es un *asynchronous iterator* el cual cuando es llamado usa el método `__anext__()` retornando un objeto a la espera (*awaitable*) el cual ejecutará el cuerpo de la función generadora asincrónica hasta la siguiente expresión `yield`.

Cada `yield` suspende temporalmente el procesamiento, recordando el estado local de ejecución (incluyendo a las variables locales y las sentencias *try* pendientes). Cuando el *iterador del generador asincrónico* vuelve efectivamente con otro objeto a la espera (*awaitable*) retornado por el método `__anext__()`, retoma donde lo dejó. Vea **PEP 492** y **PEP 525**.

iterable asincrónico Un objeto, que puede ser usado en una sentencia `async for`. Debe retornar un *asynchronous iterator* de su método `__aiter__()`. Introducido por **PEP 492**.

iterador asincrónico Un objeto que implementa los métodos `__aiter__()` y `__anext__()`. `__anext__` debe retornar un objeto *awaitable*. `async for` resuelve los esperables retornados por un método de iterador asincrónico `__anext__()` hasta que lanza una excepción `StopAsyncIteration`. Introducido por **PEP 492**.

atributo Un valor asociado a un objeto que es referencias por el nombre usado expresiones de punto. Por ejemplo, si un objeto *o* tiene un atributo *a* sería referenciado como *o.a*.

a la espera Es un objeto a la espera (*awaitable*) que puede ser usado en una expresión `await`. Puede ser una *coroutine* o un objeto con un método `__await__()`. Vea también **PEP 492**.

BDFL Sigla de *Benevolent Dictator For Life*, benevolente dictador vitalicio, es decir Guido van Rossum, el creador de Python.

archivo binario Un *file object* capaz de leer y escribir *objetos tipo binarios*. Ejemplos de archivos binarios son los abiertos en modo binario (`'rb'`, `'wb'` o `'rb+'`), `sys.stdin.buffer`, `sys.stdout.buffer`, e instancias de `io.BytesIO` y de `gzip.GzipFile`.

Vea también *text file* para un objeto archivo capaz de leer y escribir objetos `str`.

objetos tipo binarios Un objeto que soporta bufferobjects y puede exportar un búfer C-*contiguous*. Esto incluye todas los objetos `bytes`, `bytearray`, y `array.array`, así como muchos objetos comunes `memoryview`. Los objetos tipo binarios pueden ser usados para varias operaciones que usan datos binarios; éstas incluyen compresión, salvar a archivos binarios, y enviarlos a través de un socket.

Algunas operaciones necesitan que los datos binarios sean mutables. La documentación frecuentemente se refiere a éstos como «objetos tipo binario de lectura y escritura». Ejemplos de objetos de búfer mutables incluyen a `bytearray` y `memoryview` de la `bytearray`. Otras operaciones que requieren datos binarios almacenados en objetos inmutables («objetos tipo binario de sólo lectura»); ejemplos de éstos incluyen `bytes` y `memoryview` del objeto `bytes`.

bytecode El código fuente Python es compilado en *bytecode*, la representación interna de un programa python en el intérprete CPython. El *bytecode* también es guardado en caché en los archivos *.pyc* de tal forma que ejecutar el mismo archivo es más fácil la segunda vez (la recompilación desde el código fuente a *bytecode* puede ser evitada). Este «lenguaje intermedio» deberá corren en una *virtual machine* que ejecute el código de máquina correspondiente a cada *bytecode*. Note que los *bytecodes* no tienen como requisito trabajar en las diversas máquina virtuales de Python, ni de ser estable entre versiones Python.

Una lista de las instrucciones en *bytecode* está disponible en la documentación de el módulo dis.

callback A subroutine function which is passed as an argument to be executed at some point in the future.

clase Una plantilla para crear objetos definidos por el usuario. Las definiciones de clase normalmente contienen definiciones de métodos que operan una instancia de la clase.

variable de clase Una variable definida en una clase y prevista para ser modificada sólo a nivel de clase (es decir, no en una instancia de la clase).

coerción La conversión implícita de una instancia de un tipo en otra durante una operación que involucra dos argumentos del mismo tipo. Por ejemplo, `int(3.15)` convierte el número de punto flotante al entero `3`, pero en `3 + 4.5`, cada argumento es de un tipo diferente (uno entero, otro flotante), y ambos deben ser convertidos al mismo tipo antes de que puedan ser sumados o emitiría un `TypeError`. Sin coerción, todos los argumentos, incluso de tipos compatibles, deberían ser normalizados al mismo tipo por el programador, por ejemplo `float(3)+4.5` en lugar de `3+4.5`.

número complejo Una extensión del sistema familiar de número reales en el cual los números son expresados como la suma de una parte real y una parte imaginaria. Los números imaginarios son múltiplos de la unidad imaginaria (la raíz cuadrada de `-1`), usualmente escrita como `i` en matemáticas o `j` en ingeniería. Python tiene soporte incorporado para números complejos, los cuales son escritos con la notación mencionada al final.; la parte imaginaria es escrita con un sufijo `j`, por ejemplo, `3+1j`. Para tener acceso a los equivalentes complejos del módulo `math` module, use `cmath`. El uso de números complejos es matemática bastante avanzada. Si no le parecen necesarios, puede ignorarlos sin inconvenientes.

administrador de contextos Un objeto que controla el entorno en la sentencia `with` definiendo los métodos `__enter__()` y `__exit__()`. Vea **PEP 343**.

variable de contexto Una variable que puede tener diferentes valores dependiendo del contexto. Esto es similar a un almacenamiento de hilo local *Thread-Local Storage* en el cual cada hilo de ejecución puede tener valores diferentes para una variable. Sin embargo, con las variables de contexto, podría haber varios contextos en un hilo de ejecución y el uso principal de las variables de contexto es mantener registro de las variables en tareas concurrentes asíncronas. Vea `contextvars`.

contiguo Un búfer es considerado contiguo con precisión si es *C-contiguo* o *Fortran contiguo*. Los búferes cero dimensionales con C y Fortran contiguos. En los arreglos unidimensionales, los ítems deben ser dispuestos en memoria uno siguiente al otro, ordenados por índices que comienzan en cero. En arreglos unidimensionales C-contiguos, el último índice varía más velozmente en el orden de las direcciones de memoria. Sin embargo, en arreglos Fortran contiguos, el primer índice vería más rápidamente.

corrutina Las corrutinas son una forma más generalizadas de las subrutinas. A las subrutinas se ingresa por un punto y se sale por otro punto. Las corrutinas pueden se iniciadas, finalizadas y reanudadas en muchos puntos diferentes. Pueden ser implementadas con la sentencia `async def`. Vea además **PEP 492**.

función corrutina Un función que retorna un objeto *coroutine* . Una función corrutina puede ser definida con la sentencia `async def`, y puede contener las palabras claves `await`, `async for`, y `async with`. Las mismas son introducidas en **PEP 492**.

CPython La implementación canónica del lenguaje de programación Python, como se distribuye en python.org. El término «CPython» es usado cuando es necesario distinguir esta implementación de otras como *Jython* o *IronPython*.

decorador Una función que retorna otra función, usualmente aplicada como una función de transformación empleando la sintaxis `@envoltorio`. Ejemplos comunes de decoradores son `classmethod()` y `staticmethod()`.

La sintaxis del decorador es meramente azúcar sintáctico, las definiciones de las siguientes dos funciones son semánticamente equivalentes:

```
def f(...):
    ...
f = staticmethod(f)

@staticmethod
def f(...):
    ...
```

El mismo concepto existe para clases, pero son menos usadas. Vea la documentación de function definitions y class definitions para mayor detalle sobre decoradores.

descriptor Cualquier objeto que define los métodos `__get__()`, `__set__()`, o `__delete__()`. Cuando un atributo de clase es un descriptor, su conducta enlazada especial es disparada durante la búsqueda del atributo. Normalmente, usando *a.b* para consultar, establecer o borrar un atributo busca el objeto llamado *b* en el diccionario de clase de *a*, pero si *b* es un descriptor, el respectivo método descriptor es llamado. Entender descriptores es clave para lograr una comprensión profunda de Python porque son la base de muchas de las capacidades incluyendo funciones, métodos, propiedades, métodos de clase, métodos estáticos, y referencia a súper clases.

Para mayor información sobre los métodos de los descriptores vea descriptors.

diccionario Un arreglo asociativo, con claves arbitrarias que son asociadas a valores. Las claves pueden ser cualquier objeto con los métodos `__hash__()` y `__eq__()` . Son llamadas hash en Perl.

dictionary comprehension A compact way to process all or part of the elements in an iterable and return a dictionary with the results. `results = {n: n ** 2 for n in range(10)}` generates a dictionary containing key `n` mapped to value `n ** 2`. See comprehensions.

vista de diccionario Los objetos retornados por los métodos `dict.keys()`, `dict.values()`, y `dict.items()` son llamados vistas de diccionarios. Proveen una vista dinámica de las entradas de un diccionario, lo que significa que cuando el diccionario cambia, la vista refleja éstos cambios. Para forzar a la vista de diccionario a convertirse en una lista completa, use `list(dictview)`. Vea dict-views.

docstring Una cadena de caracteres literal que aparece como la primera expresión en una clase, función o módulo. Aunque es ignorada cuando se ejecuta, es reconocida por el compilador y puesta en el atributo `__doc__` de la clase, función o módulo comprendida. Como está disponible mediante introspección, es el lugar canónico para ubicar la documentación del objeto.

tipado de pato Un estilo de programación que no revisa el tipo del objeto para determinar si tiene la interfaz correcta; en vez de ello, el método o atributo es simplemente llamado o usado («Si se ve como un pato y grazna como un pato, debe ser un pato»). Enfatizando las interfaces en vez de hacerlo con los tipos específicos, un código bien diseñado pues tener mayor flexibilidad permitiendo la sustitución polimórfica. El tipado de pato *duck-typing* evita usar pruebas llamando a `type()` o `isinstance()`. (Nota: si embargo, el tipado de pato puede ser complementado con *abstract base classes*. En su lugar, generalmente pregunta con `hasattr()` o *EAFP*.

EAFP Del inglés *Easier to ask for forgiveness than permission*, es más fácil pedir perdón que pedir permiso. Este estilo de codificación común en Python asume la existencia de claves o atributos válidos y atrapa las excepciones si esta suposición resulta falsa. Este estilo rápido y limpio está caracterizado por muchas sentencias `try` y `except`. Esta técnica contrasta con estilo *LBYL* usual en otros lenguajes como C.

expresión Una construcción sintáctica que puede ser evaluada, hasta dar un valor. En otras palabras, una expresión es una acumulación de elementos de expresión tales como literales, nombres, accesos a atributos, operadores o

llamadas a funciones, todos ellos retornando valor. A diferencia de otros lenguajes, no toda la sintaxis del lenguaje son expresiones. También hay *statement*s que no pueden ser usadas como expresiones, como la `while`. Las asignaciones también son sentencias, no expresiones.

módulo de extensión Un módulo escrito en C o C++, usando la API para C de Python para interactuar con el núcleo y el código del usuario.

f-string Son llamadas *f-strings* las cadenas literales que usan el prefijo `'f'` o `'F'`, que es una abreviatura para formatted string literals. Vea también **PEP 498**.

objeto archivo Un objeto que expone una API orientada a archivos (con métodos como `read()` o `write()`) al objeto subyacente. Dependiendo de la forma en la que fue creado, un objeto archivo, puede mediar el acceso a un archivo real en el disco u otro tipo de dispositivo de almacenamiento o de comunicación (por ejemplo, entrada/salida estándar, búfer de memoria, sockets, pipes, etc.). Los objetos archivo son también denominados *objetos tipo archivo* o *flujos*.

Existen tres categorías de objetos archivo: crudos *raw archivos binarios*, con búfer *archivos binarios* y *archivos de texto*. Sus interfaces son definidas en el módulo `io`. La forma canónica de crear objetos archivo es usando la función `open()`.

objetos tipo archivo Un sinónimo de *file object*.

buscador Un objeto que trata de encontrar el *loader* para el módulo que está siendo importado.

Desde la versión 3.3 de Python, existen dos tipos de buscadores: *meta buscadores de ruta* para usar con `sys.meta_path`, y *buscadores de entradas de rutas* para usar con `sys.path_hooks`.

Vea **PEP 302**, **PEP 420** y **PEP 451** para mayores detalles.

división entera Una división matemática que se redondea hacia el entero menor más cercano. El operador de la división entera es `//`. Por ejemplo, la expresión `11 // 4` evalúa `2` a diferencia del `2.75` retornado por la verdadera división de números flotantes. Note que `(-11) // 4` es `-3` porque es `-2.75` redondeado *para abajo*. Ver **PEP 238**.

función Una serie de sentencias que retornan un valor al que las llama. También se le puede pasar cero o más *argumentos* los cuales pueden ser usados en la ejecución de la misma. Vea también *parameter*, *method*, y la sección function.

anotación de función Una *annotation* del parámetro de una función o un valor de retorno.

Las anotaciones de funciones son usadas frecuentemente para *indicadores de tipo*, por ejemplo, se espera que una función tome dos argumentos de clase `int` y también se espera que retorne dos valores `int`:

```
def sum_two_numbers(a: int, b: int) -> int:
   return a + b
```

La sintaxis de las anotaciones de funciones son explicadas en la sección function.

Vea *variable annotation* y **PEP 484**, que describen esta funcionalidad.

__future__ Un pseudo-módulo que los programadores pueden usar para habilitar nuevas capacidades del lenguaje que no son compatibles con el intérprete actual.

Al importar el módulo `__future__` y evaluar sus variables, puede verse cuándo las nuevas capacidades fueron agregadas por primera vez al lenguaje y cuando se quedaron establecidas por defecto:

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

recolección de basura El proceso de liberar la memoria de lo que ya no está en uso. Python realiza recolección de basura (*garbage collection*) llevando la cuenta de las referencias, y el recogedor de basura cíclico es capaz de detectar y romper las referencias cíclicas. El recogedor de basura puede ser controlado mediante el módulo `gc`.

generador Una función que retorna un *generator iterator*. Luce como una función normal excepto que contiene la expresión `yield` para producir series de valores utilizables en un bucle *for* o que pueden ser obtenidas una por una con la función `next()`.

Usualmente se refiere a una función generadora, pero puede referirse a un *iterador generador* en ciertos contextos. En aquellos casos en los que el significado no está claro, usar los términos completos evita la ambigüedad.

iterador generador Un objeto creado por una función *generator*.

Cada `yield` suspende temporalmente el procesamiento, recordando el estado de ejecución local (incluyendo las variables locales y las sentencias *try* pendientes). Cuando el «iterador generado» vuelve, retoma donde ha dejado, a diferencia de lo que ocurre con las funciones que comienzan nuevamente con cada invocación.

expresión generadora Una expresión que retorna un iterador. Luce como una expresión normal seguida por la cláusula `for` definiendo así una variable de bucle, un rango y una cláusula opcional `if`. La expresión combinada genera valores para la función contenedora:

```
>>> sum(i*i for i in range(10))         # sum of squares 0, 1, 4, ... 81
285
```

función genérica Una función compuesta de muchas funciones que implementan la misma operación para diferentes tipos. Qué implementación deberá ser usada durante la llamada a la misma es determinado por el algoritmo de despacho.

Vea también la entrada de glosario *single dispatch*, el decorador `functools.singledispatch()`, y **PEP 443**.

GIL Vea *global interpreter lock*.

bloqueo global del intérprete Mecanismo empleado por el intérprete *CPython* para asegurar que sólo un hilo ejecute el *bytecode* Python por vez. Esto simplifica la implementación de CPython haciendo que el modelo de objetos (incluyendo algunos críticos como `dict`) están implícitamente a salvo de acceso concurrente. Bloqueando el intérprete completo se simplifica hacerlo multi-hilos, a costa de mucho del paralelismo ofrecido por las máquinas con múltiples procesadores.

Sin embargo, algunos módulos de extensión, tanto estándar como de terceros, están diseñados para liberar el GIL cuando se realizan tareas computacionalmente intensivas como la compresión o el *hashing*. Además, el GIL siempre es liberado cuando se hace entrada/salida.

Esfuerzos previos hechos para crear un intérprete «sin hilos» (uno que bloquee los datos compartidos con una granularidad mucho más fina) no han sido exitosos debido a que el rendimiento sufrió para el caso más común de un solo procesador. Se cree que superar este problema de rendimiento haría la implementación mucho más compleja y por tanto, más costosa de mantener.

hash-based pyc Un archivo cache de *bytecode* que usa el *hash* en vez de usar el tiempo de la última modificación del archivo fuente correspondiente para determinar su validez. Vea pyc-invalidation.

hashable Un objeto es *hashable* si tiene un valor de hash que nunca cambiará durante su tiempo de vida (necesita un método `__hash__()`), y puede ser comparado con otro objeto (necesita el método `__eq__()`). Los objetos hashables que se comparan iguales deben tener el mismo número hash.

Ser *hashable* hace a un objeto utilizable como clave de un diccionario y miembro de un set, porque éstas estructuras de datos usan los valores de hash internamente.

La mayoría de los objetos inmutables incorporados en Python son *hashables*; los contenedores mutables (como las listas o los diccionarios) no lo son; los contenedores inmutables (como tuplas y conjuntos *frozensets*) son *hashables* si sus elementos son *hashables* . Los objetos que son instancias de clases definidas por el usuario son *hashables* por defecto. Todos se comparan como desiguales (excepto consigo mismos), y su valor de hash está derivado de su función `id()`.

IDLE El entorno integrado de desarrollo de Python, o *Integrated Development Environment for Python*. IDLE es un editor básico y un entorno de intérprete que se incluye con la distribución estándar de Python.

inmutable Un objeto con un valor fijo. Los objetos inmutables son números, cadenas y tuplas. Éstos objetos no pueden ser alterados. Un nuevo objeto debe ser creado si un valor diferente ha de ser guardado. Juegan un

rol importante en lugares donde es necesario un valor de hash constante, por ejemplo como claves de un diccionario.

ruta de importación Una lista de las ubicaciones (o *entradas de ruta*) que son revisadas por *path based finder* al importar módulos. Durante la importación, ésta lista de localizaciones usualmente viene de `sys.path`, pero para los subpaquetes también puede incluir al atributo `__path__` del paquete padre.

importar El proceso mediante el cual el código Python dentro de un módulo se hace alcanzable desde otro código Python en otro módulo.

importador Un objeto que buscan y lee un módulo; un objeto que es tanto *finder* como *loader*.

interactivo Python tiene un intérprete interactivo, lo que significa que puede ingresar sentencias y expresiones en el prompt del intérprete, ejecutarlos de inmediato y ver sus resultados. Sólo ejecute `python` sin argumentos (podría seleccionarlo desde el menú principal de su computadora). Es una forma muy potente de probar nuevas ideas o inspeccionar módulos y paquetes (recuerde `help(x)`).

interpretado Python es un lenguaje interpretado, a diferencia de uno compilado, a pesar de que la distinción puede ser difusa debido al compilador a *bytecode*. Esto significa que los archivos fuente pueden ser corridos directamente, sin crear explícitamente un ejecutable que es corrido luego. Los lenguajes interpretados típicamente tienen ciclos de desarrollo y depuración más cortos que los compilados, sin embargo sus programas suelen correr más lentamente. Vea también *interactive*.

apagado del intérprete Cuando se le solicita apagarse, el intérprete Python ingresa a un fase especial en la cual gradualmente libera todos los recursos reservados, como módulos y varias estructuras internas críticas. También hace varias llamadas al *recolector de basura*. Esto puede disparar la ejecución de código de destructores definidos por el usuario o *weakref callbacks*. El código ejecutado durante la fase de apagado puede encontrar varias excepciones debido a que los recursos que necesita pueden no funcionar más (ejemplos comunes son los módulos de bibliotecas o los artefactos de advertencias *warnings machinery*)

La principal razón para el apagado del intérpreter es que el módulo `__main__` o el script que estaba corriendo termine su ejecución.

iterable An object capable of returning its members one at a time. Examples of iterables include all sequence types (such as `list`, `str`, and `tuple`) and some non-sequence types like `dict`, *file objects*, and objects of any classes you define with an `__iter__()` method or with a `__getitem__()` method that implements *Sequence* semantics.

Los iterables pueden ser usados en el bucle `for` y en muchos otros sitios donde una secuencia es necesaria (`zip()`, `map()`, ...). Cuando un objeto iterable es pasado como argumento a la función incorporada `iter()`, retorna un iterador para el objeto. Este iterador pasa así el conjunto de valores. Cuando se usan iterables, normalmente no es necesario llamar a la función `iter()` o tratar con los objetos iteradores usted mismo. La sentencia `for` lo hace automáticamente por usted, creando un variable temporal sin nombre para mantener el iterador mientras dura el bucle. Vea también *iterator*, *sequence*, y *generator*.

iterador Un objeto que representa un flujo de datos. Llamadas repetidas al método `__next__()` del iterador (o al pasar la función incorporada `next()`) retorna ítems sucesivos del flujo. Cuando no hay más datos disponibles, una excepción `StopIteration` es disparada. En este momento, el objeto iterador está exhausto y cualquier llamada posterior al método `__next__()` sólo dispara otra vez `StopIteration`. Los iteradores necesitan tener un método `__iter__()` que retorna el objeto iterador mismo así cada iterador es también un iterable y puede ser usado en casi todos los lugares donde los iterables son aceptados. Una excepción importante es el código que intenta múltiples pases de iteración. Un objeto contenedor (como la `list`) produce un nuevo iterador cada vez que pasa a una función `iter()` o se usa en un bucle `for`. Intentar ésto con un iterador simplemente retornaría el mismo objeto iterador exhausto usado en previas iteraciones, haciéndolo aparecer como un contenedor vacío.

Puede encontrar más información en typeiter.

función clave Una función clave o una función de colación es un invocable que retorna un valor usado para el ordenamiento o clasificación. Por ejemplo, `locale.strxfrm()` es usada para producir claves de ordenamiento que se adaptan a las convenciones específicas de ordenamiento de un *locale*.

Cierta cantidad de herramientas de Python aceptan funciones clave para controlar como los elementos son ordenados o agrupados. Incluyendo a `min()`, `max()`, `sorted()`, `list.sort()`, `heapq.merge()`, `heapq.nsmallest()`, `heapq.nlargest()`, y `itertools.groupby()`.

Hay varias formas de crear una función clave. Por ejemplo, el método `str.lower()` puede servir como función clave para ordenamientos que no distingan mayúsculas de minúsculas. Como alternativa, una función clave puede ser realizada con una expresión `lambda` como `lambda r: (r[0], r[2])`. También, el módulo `operator` provee tres constructores de funciones clave: `attrgetter()`, `itemgetter()`, y `methodcaller()`. Vea en Sorting HOW TO ejemplos de cómo crear y usar funciones clave.

argumento nombrado Vea *argument*.

lambda Una función anónima de una línea consistente en un sola *expression* que es evaluada cuando la función es llamada. La sintaxis para crear una función lambda es `lambda [parameters]: expression`

LBYL Del inglés *Look before you leap*, «mira antes de saltar». Es un estilo de codificación que prueba explícitamente las condiciones previas antes de hacer llamadas o búsquedas. Este estilo contrasta con la manera *EAFP* y está caracterizado por la presencia de muchas sentencias `if`.

En entornos multi-hilos, el método LBYL tiene el riesgo de introducir condiciones de carrera entre los hilos que están «mirando» y los que están «saltando». Por ejemplo, el código, *if key in mapping: return mapping[key]*‘ puede fallar si otro hilo remueve *key* de *mapping* después del test, pero antes de retornar el valor. Este problema puede ser resuelto usando bloqueos o empleando el método EAFP.

lista Es una *sequence* Python incorporada. A pesar de su nombre es más similar a un arreglo en otros lenguajes que a una lista enlazada porque el acceso a los elementos es O(1).

comprensión de listas Una forma compacta de procesar todos o parte de los elementos en una secuencia y retornar una lista como resultado. `result = ['{:#04x}'.format(x) for x in range(256) if x % 2 == 0]` genera una lista de cadenas conteniendo números hexadecimales (0x..) entre 0 y 255. La cláusula `if` es opcional. Si es omitida, todos los elementos en `range(256)` son procesados.

cargador Un objeto que carga un módulo. Debe definir el método llamado `load_module()`. Un cargador es normalmente retornados por un *finder*. Vea **PEP 302** para detalles y `importlib.abc.Loader` para una *abstract base class*.

método mágico Una manera informal de llamar a un *special method*.

mapeado Un objeto contenedor que permite recupero de claves arbitrarias y que implementa los métodos especificados en la `Mapping` o `MutableMapping` abstract base classes. Por ejemplo, `dict`, `collections.defaultdict`, `collections.OrderedDict` y `collections.Counter`.

meta buscadores de ruta Un *finder* retornado por una búsqueda de `sys.meta_path`. Los meta buscadores de ruta están relacionados a *buscadores de entradas de rutas*, pero son algo diferente.

Vea en `importlib.abc.MetaPathFinder` los métodos que los meta buscadores de ruta implementan.

metaclase La clase de una clase. Las definiciones de clases crean nombres de clase, un diccionario de clase, y una lista de clases base. Las metaclases son responsables de tomar estos tres argumentos y crear la clase. La mayoría de los objetos de un lenguaje de programación orientado a objetos provienen de una implementación por defecto. Lo que hace a Python especial que es posible crear metaclases a medida. La mayoría de los usuario nunca necesitarán esta herramienta, pero cuando la necesidad surge, las metaclases pueden brindar soluciones poderosas y elegantes. Han sido usadas para *loggear* acceso de atributos, agregar seguridad a hilos, rastrear la creación de objetos, implementar *singletons*, y muchas otras tareas.

Más información hallará en metaclasses.

método Una función que es definida dentro del cuerpo de una clase. Si es llamada como un atributo de una instancia de otra clase, el método tomará el objeto instanciado como su primer *argument* (el cual es usualmente denominado *self*). Vea *function* y *nested scope*.

orden de resolución de métodos Orden de resolución de métodos es el orden en el cual una clase base es buscada por un miembro durante la búsqueda. Mire en The Python 2.3 Method Resolution Order los detalles del algoritmo usado por el intérprete Python desde la versión 2.3.

módulo Un objeto que sirve como unidad de organización del código Python. Los módulos tienen espacios de nombres conteniendo objetos Python arbitrarios. Los módulos son cargados en Python por el proceso de *importing*.

Vea también *package*.

especificador de módulo Un espacio de nombres que contiene la información relacionada a la importación usada al leer un módulo. Una instancia de `importlib.machinery.ModuleSpec`.

MRO Vea *method resolution order*.

mutable Los objetos mutables pueden cambiar su valor pero mantener su `id()`. Vea también *immutable*.

tupla nombrada La denominación «tupla nombrada» se aplica a cualquier tipo o clase que hereda de una tupla y cuyos elementos indexables son también accesibles usando atributos nombrados. Este tipo o clase puede tener además otras capacidades.

Varios tipos incorporados son tuplas nombradas, incluyendo los valores retornados por `time.localtime()` y `os.stat()`. Otro ejemplo es `sys.float_info`:

```
>>> sys.float_info[1]                    # indexed access
1024
>>> sys.float_info.max_exp               # named field access
1024
>>> isinstance(sys.float_info, tuple)    # kind of tuple
True
```

Algunas tuplas nombradas con tipos incorporados (como en los ejemplo precedentes). También puede ser creada con una definición regular de clase que hereda de la clase `tuple` y que define campos nombrados. Una clase como esta puede ser hechas personalizadamente o puede ser creada con la función factoría `collections.namedtuple()`. Esta última técnica automáticamente brinda métodos adicionales que pueden no estar presentes en las tuplas nombradas personalizadas o incorporadas.

espacio de nombres El lugar donde la variable es almacenada. Los espacios de nombres son implementados como diccionarios. Hay espacio de nombre local, global, e incorporado así como espacios de nombres anidados en objetos (en métodos). Los espacios de nombres soportan modularidad previniendo conflictos de nombramiento. Por ejemplo, las funciones `builtins.open` y `os.open()` se distinguen por su espacio de nombres. Los espacios de nombres también ayuda a la legibilidad y mantenibilidad dejando claro qué módulo implementa una función. Por ejemplo, escribiendo `random.seed()` o `itertools.islice()` queda claro que éstas funciones están implementadas en los módulos `random` y `itertools`, respectivamente.

paquete de espacios de nombres Un **PEP 420** *package* que sirve sólo para contener subpaquetes. Los paquetes de espacios de nombres pueden no tener representación física, y específicamente se diferencian de los *regular package* porque no tienen un archivo `__init__.py`.

Vea también *module*.

alcances anidados La habilidad de referirse a una variable dentro de una definición encerrada. Por ejemplo, una función definida dentro de otra función puede referir a variables en la función externa. Note que los alcances anidados por defecto sólo funcionan para referencia y no para asignación. Las variables locales leen y escriben sólo en el alcance más interno. De manera semejante, las variables globales pueden leer y escribir en el espacio de nombres global. Con `nonlocal` se puede escribir en alcances exteriores.

clase de nuevo estilo Vieja denominación usada para el estilo de clases ahora empleado en todos los objetos de clase. En versiones más tempranas de Python, sólo las nuevas clases podían usar capacidades nuevas y versátiles de Python como `__slots__`, descriptores, propiedades, `__getattribute__()`, métodos de clase y métodos estáticos.

objeto Cualquier dato con estado (atributo o valor) y comportamiento definido (métodos). También es la más básica clase base para cualquier *new-style class*.

paquete Un *module* Python que puede contener submódulos o recursivamente, subpaquetes. Técnicamente, un paquete es un módulo Python con un atributo `__path__`.

Vea también *regular package* y *namespace package*.

parámetro Una entidad nombrada en una definición de una *function* (o método) que especifica un *argument* (o en algunos casos, varios argumentos) que la función puede aceptar. Existen cinco tipos de argumentos:

- *posicional o nombrado*: especifica un argumento que puede ser pasado tanto como *posicional* o como *nombrado*. Este es el tipo por defecto de parámetro, como *foo* y *bar* en el siguiente ejemplo:

```
def func(foo, bar=None): ...
```

- *sólo posicional*: especifica un argumento que puede ser pasado sólo por posición. Los parámetros sólo posicionales pueden ser definidos incluyendo un carácter / en la lista de parámetros de la función después de ellos, como *posonly1* y *posonly2* en el ejemplo que sigue:

```
def func(posonly1, posonly2, /, positional_or_keyword): ...
```

- *sólo nombrado*: especifica un argumento que sólo puede ser pasado por nombre. Los parámetros sólo por nombre pueden ser definidos incluyendo un parámetro posicional de una sola variable o un simple *` antes de ellos en la lista de parámetros en la definición de la función, como *kw_only1* y *kw_only2* en el ejemplo siguiente:

```
def func(arg, *, kw_only1, kw_only2): ...
```

- *variable posicional*: especifica una secuencia arbitraria de argumentos posicionales que pueden ser brindados (además de cualquier argumento posicional aceptado por otros parámetros). Este parámetro puede ser definido anteponiendo al nombre del parámetro *, como a *args* en el siguiente ejemplo:

```
def func(*args, **kwargs): ...
```

- *variable nombrado*: especifica que arbitrariamente muchos argumentos nombrados pueden ser brindados (además de cualquier argumento nombrado ya aceptado por cualquier otro parámetro). Este parámetro puede ser definido anteponiendo al nombre del parámetro con **, como *kwargs* en el ejemplo precedente.

Los parámetros puede especificar tanto argumentos opcionales como requeridos, así como valores por defecto para algunos argumentos opcionales.

Vea también el glosario de *argument*, la pregunta respondida en *la diferencia entre argumentos y parámetros*, la clase `inspect.Parameter`, la sección function , y **PEP 362**.

entrada de ruta Una ubicación única en el *import path* que el *path based finder* consulta para encontrar los módulos a importar.

buscador de entradas de ruta Un *finder* retornado por un invocable en `sys.path_hooks` (esto es, un *path entry hook*) que sabe cómo localizar módulos dada una *path entry*.

Vea en `importlib.abc.PathEntryFinder` los métodos que los buscadores de entradas de ruta implementan.

gancho a entrada de ruta Un invocable en la lista `sys.path_hook` que retorna un *path entry finder* si éste sabe cómo encontrar módulos en un *path entry* específico.

buscador basado en ruta Uno de los *meta buscadores de ruta* por defecto que busca un *import path* para los módulos.

objeto tipo ruta Un objeto que representa una ruta del sistema de archivos. Un objeto tipo ruta puede ser tanto una `str` como un `bytes` representando una ruta, o un objeto que implementa el protocolo `os.PathLike`. Un objeto que soporta el protocolo `os.PathLike` puede ser convertido a ruta del sistema de archivo de clase `str` o `bytes` usando la función `os.fspath()`; `os.fsdecode()` `os.fsencode()` pueden emplearse para garantizar que retorne respectivamente `str` o `bytes`. Introducido por **PEP 519**.

PEP Propuesta de mejora de Python, del inglés *Python Enhancement Proposal*. Un PEP es un documento de diseño que brinda información a la comunidad Python, o describe una nueva capacidad para Python, sus procesos o entorno. Los PEPs deberían dar una especificación técnica concisa y una fundamentación para las capacidades propuestas.

Los PEPs tienen como propósito ser los mecanismos primarios para proponer nuevas y mayores capacidad, para recoger la opinión de la comunidad sobre un tema, y para documentar las decisiones de diseño que se han hecho en Python. El autor del PEP es el responsable de lograr consenso con la comunidad y documentar las opiniones disidentes.

Vea **PEP 1**.

porción Un conjunto de archivos en un único directorio (posiblemente guardo en un archivo comprimido *zip*) que contribuye a un espacio de nombres de paquete, como está definido en **PEP 420**.

argumento posicional Vea *argument*.

API provisoria Una API provisoria es aquella que deliberadamente fue excluida de las garantías de compatibilidad hacia atrás de la biblioteca estándar. Aunque no se esperan cambios fundamentales en dichas interfaces, como están marcadas como provisionales, los cambios incompatibles hacia atrás (incluso remover la misma interfaz) podrían ocurrir si los desarrolladores principales lo estiman. Estos cambios no se hacen gratuitamente – solo ocurrirán si fallas fundamentales y serias son descubiertas que no fueron vistas antes de la inclusión de la API.

Incluso para APIs provisorias, los cambios incompatibles hacia atrás son vistos como una «solución de último recurso» - se intentará todo para encontrar una solución compatible hacia atrás para los problemas identificados.

Este proceso permite que la biblioteca estándar continúe evolucionando con el tiempo, sin bloquearse por errores de diseño problemáticos por períodos extensos de tiempo. Vea **PEP 411** para más detalles.

paquete provisorio Vea *provisional API*.

Python 3000 Apodo para la fecha de lanzamiento de Python 3.x (acuñada en un tiempo cuando llegar a la versión 3 era algo distante en el futuro.) También se lo abrevió como *Py3k*.

Pythónico Una idea o pieza de código que sigue ajustadamente la convenciones idiomáticas comunes del lenguaje Python, en vez de implementar código usando conceptos comunes a otros lenguajes. Por ejemplo, una convención común en Python es hacer bucles sobre todos los elementos de un iterable con la sentencia `for`. Muchos otros lenguajes no tienen este tipo de construcción, así que los que no están familiarizados con Python podrían usar contadores numéricos:

```
for i in range(len(food)):
    print(food[i])
```

En contraste, un método Pythónico más limpio:

```
for piece in food:
    print(piece)
```

nombre calificado Un nombre con puntos mostrando la ruta desde el alcance global del módulo a la clase, función o método definido en dicho módulo, como se define en **PEP 3155**. Para las funciones o clases de más alto nivel, el nombre calificado es el igual al nombre del objeto:

```
>>> class C:
...     class D:
...         def meth(self):
...             pass
...
>>> C.__qualname__
'C'
>>> C.D.__qualname__
'C.D'
>>> C.D.meth.__qualname__
'C.D.meth'
```

Cuando es usado para referirse a los módulos, *nombre completamente calificado* significa la ruta con puntos completo al módulo, incluyendo cualquier paquete padre, por ejemplo, *email.mime.text*‘:

```
>>> import email.mime.text
>>> email.mime.text.__name__
'email.mime.text'
```

contador de referencias El número de referencias a un objeto. Cuando el contador de referencias de un objeto cae hasta cero, éste es desalojable. En conteo de referencias no suele ser visible en el código de Python, pero es un elemento clave para la implementación de *CPython*. El módulo `sys` define la `getrefcount()` que los programadores pueden emplear para retornar el conteo de referencias de un objeto en particular.

paquete regular Un *package* tradicional, como aquellos con un directorio conteniendo el archivo `__init__.py`.

Vea también *namespace package*.

__slots__ Es una declaración dentro de una clase que ahorra memoria predeclarando espacio para las atributos de la instancia y eliminando diccionarios de la instancia. Aunque es popular, esta técnica es algo dificultosa de lograr correctamente y es mejor reservarla para los casos raros en los que existen grandes cantidades de instancias en aplicaciones con uso crítico de memoria.

secuencia Un *iterable* que logra un acceso eficiente a los elementos usando índices enteros a través del método especial `__getitem__()` y que define un método `__len__()` que retorna la longitud de la secuencia. Algunas de las secuencias incorporadas son `list`, `str`, `tuple`, y `bytes`. Observe que `dict` también soporta `__getitem__()` y `__len__()`, pero es considerada un mapeo más que una secuencia porque las búsquedas son por claves arbitraria *immutable* y no por enteros.

The `collections.abc.Sequence` abstract base class defines a much richer interface that goes beyond just `__getitem__()` and `__len__()`, adding `count()`, `index()`, `__contains__()`, and `__reversed__()`. Types that implement this expanded interface can be registered explicitly using `register()`.

set comprehension A compact way to process all or part of the elements in an iterable and return a set with the results. `results = {c for c in 'abracadabra' if c not in 'abc'}` generates the set of strings `{'r', 'd'}`. See comprehensions.

despacho único Una forma de despacho de una *generic function* donde la implementación es elegida a partir del tipo de un sólo argumento.

rebanada Un objeto que contiene una porción de una *sequence*. Una rebanada es creada usando la notación de suscripto, `[]` con dos puntos entre los números cuando se ponen varios, como en `nombre_variable[1:3:5]`. La notación con corchete (suscrito) usa internamente objetos `slice`.

método especial Un método que es llamado implícitamente por Python cuando ejecuta ciertas operaciones en un tipo, como la adición. Estos métodos tienen nombres que comienzan y terminan con doble barra baja. Los métodos especiales están documentados en specialnames.

sentencia Una sentencia es parte de un conjunto (un «bloque» de código). Una sentencia tanto es una *expression* como alguna de las varias sintaxis usando una palabra clave, como `if`, `while` o `for`.

codificación de texto Un códec que codifica las cadenas Unicode a bytes.

archivo de texto Un *file object* capaz de leer y escribir objetos `str`. Frecuentemente, un archivo de texto también accede a un flujo de datos binario y maneja automáticamente el *text encoding*. Ejemplos de archivos de texto que son abiertos en modo texto (`'r'` o `'w'`), `sys.stdin`, `sys.stdout`, y las instancias de `io.StringIO`.

Vea también *binary file* por objeto de archivos capaces de leer y escribir *objeto tipo binario*.

cadena con triple comilla Una cadena que está enmarcada por tres instancias de comillas (») o apostrofes (‘). Aunque no brindan ninguna funcionalidad que no está disponible usando cadenas con comillas simple, son útiles por varias razones. Permiten incluir comillas simples o dobles sin escapar dentro de las cadenas y pueden abarcar múltiples líneas sin el uso de caracteres de continuación, haciéndolas particularmente útiles para escribir docstrings.

tipo El tipo de un objeto Python determina qué tipo de objeto es; cada objeto tiene un tipo. El tipo de un objeto puede ser accedido por su atributo `__class__` o puede ser conseguido usando `type(obj)`.

alias de tipos Un sinónimo para un tipo, creado al asignar un tipo a un identificador.

Los alias de tipos son útiles para simplificar los *indicadores de tipo*. Por ejemplo:

```
from typing import List, Tuple

def remove_gray_shades(
        colors: List[Tuple[int, int, int]]) -> List[Tuple[int, int, int]]:
    pass
```

podría ser más legible así:

```
from typing import List, Tuple

Color = Tuple[int, int, int]

def remove_gray_shades(colors: List[Color]) -> List[Color]:
    pass
```

Vea `typing` y **PEP 484**, que describen esta funcionalidad.

indicador de tipo Una *annotation* que especifica el tipo esperado para una variable, un atributo de clase, un parámetro para una función o un valor de retorno.

Los indicadores de tipo son opcionales y no son obligados por Python pero son útiles para las herramientas de análisis de tipos estático, y ayuda a las IDE en el completado del código y la refactorización.

Los indicadores de tipo de las variables globales, atributos de clase, y funciones, no de variables locales, pueden ser accedidos usando `typing.get_type_hints()`.

Vea `typing` y **PEP 484**, que describen esta funcionalidad.

saltos de líneas universales Una manera de interpretar flujos de texto en la cual son reconocidos como finales de línea todas siguientes formas: la convención de Unix para fin de línea `'\n'`, la convención de Windows `'\r\n'`, y la vieja convención de Macintosh `'\r'`. Vea **PEP 278** y **PEP 3116**, además de `bytes.splitlines()` para usos adicionales.

anotación de variable Una *annotation* de una variable o un atributo de clase.

Cuando se anota una variable o un atributo de clase, la asignación es opcional:

```
class C:
    field: 'annotation'
```

Las anotaciones de variables son frecuentemente usadas para *type hints*: por ejemplo, se espera que esta variable tenga valores de clase `int`:

```
count: int = 0
```

La sintaxis de la anotación de variables está explicada en la sección annassign.

Vea *function annotation*, **PEP 484** y **PEP 526**, los cuales describen esta funcionalidad.

entorno virtual Un entorno cooperativamente aislado de ejecución que permite a los usuarios de Python y a las aplicaciones instalar y actualizar paquetes de distribución de Python sin interferir con el comportamiento de otras aplicaciones de Python en el mismo sistema.

Vea también `venv`.

máquina virtual Una computadora definida enteramente por software. La máquina virtual de Python ejecuta el *bytecode* generado por el compilador de *bytecode*.

Zen de Python Un listado de los principios de diseño y la filosofía de Python que son útiles para entender y usar el lenguaje. El listado puede encontrarse ingresando «`import this`» en la consola interactiva.

APÉNDICE B

Acerca de estos documentos

Estos documentos son generados por reStructuredText desarrollado por Sphinx, un procesador de documentos específicamente escrito para la documentación de Python.

El desarrollo de la documentación y su cadena de herramientas es un esfuerzo enteramente voluntario, al igual que Python. Si tu quieres contribuir, por favor revisa la página reporting-bugs para más información de cómo hacerlo. Los nuevos voluntarios son siempre bienvenidos!

Agradecemos a:

- Fred L. Drake, Jr., el creador original de la documentación del conjunto de herramientas de Python y escritor de gran parte del contenido;
- el proyecto Docutils para creación de reStructuredText y el juego de Utilidades de Documentación;
- Fredrik Lundh por su proyecto Referencia Alternativa de Python para la cual Sphinx tuvo muchas ideas.

B.1 Contribuidores de la documentación de Python

Muchas personas han contribuido para el lenguaje de Python, la librería estándar de Python, y la documentación de Python. Revisa Misc/ACKS la distribución de Python para una lista parcial de contribuidores.

Es solamente con la aportación y contribuciones de la comunidad de Python que Python tiene tan fantástica documentación – Muchas gracias!

APÉNDICE C

History and License

C.1 History of the software

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI, see https://www.cwi.nl/) in the Netherlands as a successor of a language called ABC. Guido remains Python's principal author, although it includes many contributions from others.

In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI, see https://www.cnri.reston.va.us/) in Reston, Virginia where he released several versions of the software.

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. In October of the same year, the PythonLabs team moved to Digital Creations (now Zope Corporation; see https://www.zope.org/). In 2001, the Python Software Foundation (PSF, see https://www.python.org/psf/) was formed, a non-profit organization created specifically to own Python-related Intellectual Property. Zope Corporation is a sponsoring member of the PSF.

All Python releases are Open Source (see https://opensource.org/ for the Open Source Definition). Historically, most, but not all, Python releases have also been GPL-compatible; the table below summarizes the various releases.

Release	Derived from	Year	Owner	GPL compatible?
0.9.0 thru 1.2	n/a	1991-1995	CWI	yes
1.3 thru 1.5.2	1.2	1995-1999	CNRI	yes
1.6	1.5.2	2000	CNRI	no
2.0	1.6	2000	BeOpen.com	no
1.6.1	1.6	2001	CNRI	no
2.1	2.0+1.6.1	2001	PSF	no
2.0.1	2.0+1.6.1	2001	PSF	yes
2.1.1	2.1+2.0.1	2001	PSF	yes
2.1.2	2.1.1	2002	PSF	yes
2.1.3	2.1.2	2002	PSF	yes
2.2 and above	2.1.1	2001-now	PSF	yes

Nota: GPL-compatible doesn't mean that we're distributing Python under the GPL. All Python licenses, unlike the GPL, let you distribute a modified version without making your changes open source. The GPL-compatible licenses make it possible to combine Python with other software that is released under the GPL; the others don't.

Thanks to the many outside volunteers who have worked under Guido's direction to make these releases possible.

C.2 Terms and conditions for accessing or otherwise using Python

Python software and documentation are licensed under the *PSF License Agreement*.

Starting with Python 3.8.6, examples, recipes, and other code in the documentation are dual licensed under the PSF License Agreement and the *Zero-Clause BSD license*.

Some software incorporated into Python is under different licenses. The licenses are listed with code falling under that license. See *Licenses and Acknowledgements for Incorporated Software* for an incomplete list of these licenses.

C.2.1 PSF LICENSE AGREEMENT FOR PYTHON 3.8.20

```
1. This LICENSE AGREEMENT is between the Python Software Foundation␣
↪("PSF"), and
   the Individual or Organization ("Licensee") accessing and otherwise␣
↪using Python
   3.8.20 software in source or binary form and its associated␣
↪documentation.

2. Subject to the terms and conditions of this License Agreement, PSF␣
↪hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to␣
↪reproduce,
   analyze, test, perform and/or display publicly, prepare derivative␣
↪works,
   distribute, and otherwise use Python 3.8.20 alone or in any derivative
   version, provided, however, that PSF's License Agreement and PSF's␣
↪notice of
   copyright, i.e., "Copyright © 2001-2023 Python Software Foundation; All␣
↪Rights
   Reserved" are retained in Python 3.8.20 alone or in any derivative␣
↪version
   prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python 3.8.20 or any part thereof, and wants to make the
   derivative work available to others as provided herein, then Licensee␣
↪hereby
   agrees to include in any such work a brief summary of the changes made␣
↪to Python
   3.8.20.

4. PSF is making Python 3.8.20 available to Licensee on an "AS IS" basis.
   PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY␣
↪OF
   EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY␣
↪REPRESENTATION OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR␣
↪THAT THE
   USE OF PYTHON 3.8.20 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.8.20
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A␣
↪RESULT OF
```

```
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.8.20, OR ANY␣
↪DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material␣
↪breach of
   its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any␣
↪relationship
   of agency, partnership, or joint venture between PSF and Licensee. ␣
↪This License
   Agreement does not grant permission to use PSF trademarks or trade name␣
↪in a
   trademark sense to endorse or promote products or services of Licensee,␣
↪or any
   third party.

8. By copying, installing or otherwise using Python 3.8.20, Licensee agrees
   to be bound by the terms and conditions of this License Agreement.
```

C.2.2 BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0

BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1

```
1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at
   160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization
   ("Licensee") accessing and otherwise using this software in source or binary
   form and its associated documentation ("the Software").

2. Subject to the terms and conditions of this BeOpen Python License Agreement,
   BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license
   to reproduce, analyze, test, perform and/or display publicly, prepare derivative
   works, distribute, and otherwise use the Software alone or in any derivative
   version, provided, however, that the BeOpen Python License is retained in the
   Software, alone or in any derivative version prepared by Licensee.

3. BeOpen is making the Software available to Licensee on an "AS IS" basis.
   BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF
   EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE
   USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING,
   MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF
   ADVISED OF THE POSSIBILITY THEREOF.

5. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

6. This License Agreement shall be governed by and interpreted in all respects
   by the law of the State of California, excluding conflict of law provisions.
   Nothing in this License Agreement shall be deemed to create any relationship of
   agency, partnership, or joint venture between BeOpen and Licensee.  This License
   Agreement does not grant permission to use BeOpen trademarks or trade names in a
   trademark sense to endorse or promote products or services of Licensee, or any
   third party.  As an exception, the "BeOpen Python" logos available at
   http://www.pythonlabs.com/logos.html may be used according to the permissions
```

(continué en la próxima página)

(proviene de la página anterior)

```
   granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be
   bound by the terms and conditions of this License Agreement.
```

C.2.3 CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1

```
1. This LICENSE AGREEMENT is between the Corporation for National Research
   Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191
   ("CNRI"), and the Individual or Organization ("Licensee") accessing and
   otherwise using Python 1.6.1 software in source or binary form and its
   associated documentation.

2. Subject to the terms and conditions of this License Agreement, CNRI hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python 1.6.1 alone or in any derivative version,
   provided, however, that CNRI's License Agreement and CNRI's notice of copyright,
   i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All
   Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version
   prepared by Licensee.  Alternately, in lieu of CNRI's License Agreement,
   Licensee may substitute the following text (omitting the quotes): "Python 1.6.1
   is made available subject to the terms and conditions in CNRI's License
   Agreement.  This Agreement together with Python 1.6.1 may be located on the
   Internet using the following unique, persistent identifier (known as a handle):
   1895.22/1013.  This Agreement may also be obtained from a proxy server on the
   Internet using the following URL: http://hdl.handle.net/1895.22/1013."

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python 1.6.1 or any part thereof, and wants to make the derivative
   work available to others as provided herein, then Licensee hereby agrees to
   include in any such work a brief summary of the changes made to Python 1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis.  CNRI
   MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF EXAMPLE,
   BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY
   OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF
   PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

7. This License Agreement shall be governed by the federal intellectual property
   law of the United States, including without limitation the federal copyright
   law, and, to the extent such U.S. federal law does not apply, by the law of the
   Commonwealth of Virginia, excluding Virginia's conflict of law provisions.
   Notwithstanding the foregoing, with regard to derivative works based on Python
   1.6.1 that incorporate non-separable material that was previously distributed
   under the GNU General Public License (GPL), the law of the Commonwealth of
   Virginia shall govern this License Agreement only as to issues arising under or
   with respect to Paragraphs 4, 5, and 7 of this License Agreement.  Nothing in
   this License Agreement shall be deemed to create any relationship of agency,
   partnership, or joint venture between CNRI and Licensee.  This License Agreement
   does not grant permission to use CNRI trademarks or trade name in a trademark
```

(continué en la próxima página)

(proviene de la página anterior)

```
   sense to endorse or promote products or services of Licensee, or any third
   party.

8. By clicking on the "ACCEPT" button where indicated, or by copying, installing
   or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and
   conditions of this License Agreement.
```

C.2.4 CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2

```
Copyright © 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The
Netherlands.  All rights reserved.

Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted, provided that
the above copyright notice appear in all copies and that both that copyright
notice and this permission notice appear in supporting documentation, and that
the name of Stichting Mathematisch Centrum or CWI not be used in advertising or
publicity pertaining to distribution of the software without specific, written
prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO
EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT
OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE,
DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS
SOFTWARE.
```

C.2.5 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON 3.8.20 DOCUMENTATION

```
Permission to use, copy, modify, and/or distribute this software for any
purpose with or without fee is hereby granted.

THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH
REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT,
INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM
LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3 Licenses and Acknowledgements for Incorporated Software

This section is an incomplete, but growing list of licenses and acknowledgements for third-party software incorporated in the Python distribution.

C.3.1 Mersenne Twister

The `_random` module includes code based on a download from http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html. The following are the verbatim comments from the original code:

```
A C-program for MT19937, with initialization improved 2002/1/26.
Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using init_genrand(seed)
or init_by_array(init_key, key_length).

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

 1. Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

 2. Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in the
    documentation and/or other materials provided with the distribution.

 3. The names of its contributors may not be used to endorse or promote
    products derived from this software without specific prior written
    permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.


Any feedback is very welcome.
http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)
```

C.3.2 Sockets

The `socket` module uses the functions, `getaddrinfo()`, and `getnameinfo()`, which are coded in separate source files from the WIDE Project, http://www.wide.ad.jp/.

```
Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
```

(continué en la próxima página)

(proviene de la página anterior)

```
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors
   may be used to endorse or promote products derived from this software
   without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.3 Asynchronous socket services

The `asynchat` and `asyncore` modules contain the following notice:

```
Copyright 1996 by Sam Rushing

                        All Rights Reserved

Permission to use, copy, modify, and distribute this software and
its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of Sam
Rushing not be used in advertising or publicity pertaining to
distribution of the software without specific, written prior
permission.

SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN
NO EVENT SHALL SAM RUSHING BE LIABLE FOR ANY SPECIAL, INDIRECT OR
CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,
NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN
CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

C.3.4 Cookie management

The `http.cookies` module contains the following notice:

```
Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

               All Rights Reserved

Permission to use, copy, modify, and distribute this software
and its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Timothy O'Malley  not be used in advertising or publicity
pertaining to distribution of the software without specific, written
```

(continué en la próxima página)

(proviene de la página anterior)

```
prior permission.

Timothy O'Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE FOR
ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3.5 Execution tracing

The `trace` module contains the following notice:

```
portions copyright 2001, Autonomous Zones Industries, Inc., all rights...
err...  reserved and offered to the public under the terms of the
Python 2.2 license.
Author: Zooko O'Whielacronx
http://zooko.com/
mailto:zooko@zooko.com

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights reserved.


Permission to use, copy, modify, and distribute this Python software and
its associated documentation for any purpose without fee is hereby
granted, provided that the above copyright notice appears in all copies,
and that both that copyright notice and this permission notice appear in
supporting documentation, and that the name of neither Automatrix,
Bioreason or Mojam Media be used in advertising or publicity pertaining to
distribution of the software without specific, written prior permission.
```

C.3.6 UUencode and UUdecode functions

The `uu` module contains the following notice:

```
Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
                       All Rights Reserved
Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Lance Ellinghouse
not be used in advertising or publicity pertaining to distribution
of the software without specific, written prior permission.
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
```

(continué en la próxima página)

(proviene de la página anterior)

```
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup. The C
  version is still 5 times faster, though.
- Arguments more compliant with Python standard
```

C.3.7 XML Remote Procedure Calls

The `xmlrpc.client` module contains the following notice:

```
    The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB
Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its
associated documentation, you agree that you have read, understood,
and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software and
its associated documentation for any purpose and without fee is
hereby granted, provided that the above copyright notice appears in
all copies, and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Secret Labs AB or the author not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD
TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANT-
ABILITY AND FITNESS.  IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR
BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE
OF THIS SOFTWARE.
```

C.3.8 test_epoll

The `test_epoll` module contains the following notice:

```
Copyright (c) 2001-2006 Twisted Matrix Laboratories.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
```

(continué en la próxima página)

(proviene de la página anterior)

```
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.9 Select kqueue

The `select` module contains the following notice for the kqueue interface:

```
Copyright (c) 2000 Doug White, 2006 James Knight, 2007 Christian Heimes
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.10 SipHash24

The file `Python/pyhash.c` contains Marek Majkowski" implementation of Dan Bernstein's SipHash24 algorithm. It contains the following note:

```
<MIT License>
Copyright (c) 2013  Marek Majkowski <marek@popcount.org>

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.
</MIT License>
```

(continué en la próxima página)

(proviene de la página anterior)

```
Original location:
   https://github.com/majek/csiphash/

Solution inspired by code from:
   Samuel Neves (supercop/crypto_auth/siphash24/little)
   djb (supercop/crypto_auth/siphash24/little2)
   Jean-Philippe Aumasson (https://131002.net/siphash/siphash24.c)
```

C.3.11 strtod and dtoa

The file `Python/dtoa.c`, which supplies C functions dtoa and strtod for conversion of C doubles to and from strings, is derived from the file of the same name by David M. Gay, currently available from http://www.netlib.org/fp/. The original file, as retrieved on March 16, 2009, contains the following copyright and licensing notice:

```
/****************************************************************
 *
 * The author of this software is David M. Gay.
 *
 * Copyright (c) 1991, 2000, 2001 by Lucent Technologies.
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose without fee is hereby granted, provided that this entire notice
 * is included in all copies of any software which is or includes a copy
 * or modification of this software and in all copies of the supporting
 * documentation for such software.
 *
 * THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED
 * WARRANTY.  IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY
 * REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY
 * OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
 *
 ***************************************************************/
```

C.3.12 OpenSSL

The modules `hashlib`, `posix`, `ssl`, `crypt` use the OpenSSL library for added performance if made available by the operating system. Additionally, the Windows and Mac OS X installers for Python may include a copy of the OpenSSL libraries, so we include a copy of the OpenSSL license here:

```
 LICENSE ISSUES
 ==============

 The OpenSSL toolkit stays under a dual license, i.e. both the conditions of
 the OpenSSL License and the original SSLeay license apply to the toolkit.
 See below for the actual license texts. Actually both licenses are BSD-style
 Open Source licenses. In case of any license issues related to OpenSSL
 please contact openssl-core@openssl.org.

 OpenSSL License
 ---------------

   /* ====================================================================
    * Copyright (c) 1998-2008 The OpenSSL Project.  All rights reserved.
    *
    * Redistribution and use in source and binary forms, with or without
    * modification, are permitted provided that the following conditions
    * are met:
```

(continué en la próxima página)

(proviene de la página anterior)

```
    *
    * 1. Redistributions of source code must retain the above copyright
    *    notice, this list of conditions and the following disclaimer.
    *
    * 2. Redistributions in binary form must reproduce the above copyright
    *    notice, this list of conditions and the following disclaimer in
    *    the documentation and/or other materials provided with the
    *    distribution.
    *
    * 3. All advertising materials mentioning features or use of this
    *    software must display the following acknowledgment:
    *    "This product includes software developed by the OpenSSL Project
    *    for use in the OpenSSL Toolkit. (http://www.openssl.org/)"
    *
    * 4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to
    *    endorse or promote products derived from this software without
    *    prior written permission. For written permission, please contact
    *    openssl-core@openssl.org.
    *
    * 5. Products derived from this software may not be called "OpenSSL"
    *    nor may "OpenSSL" appear in their names without prior written
    *    permission of the OpenSSL Project.
    *
    * 6. Redistributions of any form whatsoever must retain the following
    *    acknowledgment:
    *    "This product includes software developed by the OpenSSL Project
    *    for use in the OpenSSL Toolkit (http://www.openssl.org/)"
    *
    * THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT ``AS IS'' AND ANY
    * EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
    * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
    * PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE OpenSSL PROJECT OR
    * ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
    * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
    * NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
    * LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
    * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
    * STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
    * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
    * OF THE POSSIBILITY OF SUCH DAMAGE.
    * ====================================================================
    *
    * This product includes cryptographic software written by Eric Young
    * (eay@cryptsoft.com).  This product includes software written by Tim
    * Hudson (tjh@cryptsoft.com).
    *
    */

Original SSLeay License
-----------------------

   /* Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com)
    * All rights reserved.
    *
    * This package is an SSL implementation written
    * by Eric Young (eay@cryptsoft.com).
    * The implementation was written so as to conform with Netscapes SSL.
    *
    * This library is free for commercial and non-commercial use as long as
    * the following conditions are aheared to.  The following conditions
    * apply to all code found in this distribution, be it the RC4, RSA,
```

(continué en la próxima página)

(proviene de la página anterior)

```
    * lhash, DES, etc., code; not just the SSL code.  The SSL documentation
    * included with this distribution is covered by the same copyright terms
    * except that the holder is Tim Hudson (tjh@cryptsoft.com).
    *
    * Copyright remains Eric Young's, and as such any Copyright notices in
    * the code are not to be removed.
    * If this package is used in a product, Eric Young should be given attribution
    * as the author of the parts of the library used.
    * This can be in the form of a textual message at program startup or
    * in documentation (online or textual) provided with the package.
    *
    * Redistribution and use in source and binary forms, with or without
    * modification, are permitted provided that the following conditions
    * are met:
    * 1. Redistributions of source code must retain the copyright
    *    notice, this list of conditions and the following disclaimer.
    * 2. Redistributions in binary form must reproduce the above copyright
    *    notice, this list of conditions and the following disclaimer in the
    *    documentation and/or other materials provided with the distribution.
    * 3. All advertising materials mentioning features or use of this software
    *    must display the following acknowledgement:
    *    "This product includes cryptographic software written by
    *     Eric Young (eay@cryptsoft.com)"
    *    The word 'cryptographic' can be left out if the rouines from the library
    *    being used are not cryptographic related :-).
    * 4. If you include any Windows specific code (or a derivative thereof) from
    *    the apps directory (application code) you must include an acknowledgement:
    *    "This product includes software written by Tim Hudson (tjh@cryptsoft.com)"
    *
    * THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ``AS IS'' AND
    * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
    * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
    * ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
    * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
    * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
    * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
    * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
    * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
    * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
    * SUCH DAMAGE.
    *
    * The licence and distribution terms for any publically available version or
    * derivative of this code cannot be changed.  i.e. this code cannot simply be
    * copied and put under another distribution licence
    * [including the GNU Public Licence.]
    */
```

C.3.13 expat

The `pyexpat` extension is built using an included copy of the expat sources unless the build is configured `--with-system-expat`:

```
Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd
                               and Clark Cooper

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
```

(continué en la próxima página)

(proviene de la página anterior)

```
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.14 libffi

The `_ctypes` extension is built using an included copy of the libffi sources unless the build is configured `--with-system-libffi`:

```
Copyright (c) 1996-2008  Red Hat, Inc and others.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
``Software''), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED ``AS IS'', WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT.  IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```

C.3.15 zlib

The `zlib` extension is built using an included copy of the zlib sources if the zlib version found on the system is too old to be used for the build:

```
Copyright (C) 1995-2011 Jean-loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied
warranty.  In no event will the authors be held liable for any damages
arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:
```

(continué en la próxima página)

(proviene de la página anterior)

```
1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would be
   appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.

3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly        Mark Adler
jloup@gzip.org          madler@alumni.caltech.edu
```

C.3.16 cfuhash

The implementation of the hash table used by the `tracemalloc` is based on the cfuhash project:

```
Copyright (c) 2005 Don Owens
All rights reserved.

This code is released under the BSD license:

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

  * Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

  * Redistributions in binary form must reproduce the above
    copyright notice, this list of conditions and the following
    disclaimer in the documentation and/or other materials provided
    with the distribution.

  * Neither the name of the author nor the names of its
    contributors may be used to endorse or promote products derived
    from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
OF THE POSSIBILITY OF SUCH DAMAGE.
```

C.3.17 libmpdec

The `_decimal` module is built using an included copy of the libmpdec library unless the build is configured `--with-system-libmpdec`:

```
Copyright (c) 2008-2016 Stefan Krah. All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.18 W3C C14N test suite

The C14N 2.0 test suite in the `test` package (`Lib/test/xmltestdata/c14n-20/`) was retrieved from the W3C website at https://www.w3.org/TR/xml-c14n2-testcases/ and is distributed under the 3-clause BSD license:

> Copyright (c) 2013 W3C(R) (MIT, ERCIM, Keio, Beihang), All Rights Reserved.
>
> Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:
>
> - Redistributions of works must retain the original copyright notice, this list of conditions and the following disclaimer.
> - Redistributions in binary form must reproduce the original copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
> - Neither the name of the W3C nor the names of its contributors may be used to endorse or promote products derived from this work without specific prior written permission.
>
> THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS «AS IS» AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

APÉNDICE D

Copyright

See *History and License* for complete license and permissions information.

Índice

No alfabético

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

V

Z