Cleisson de Oliveira Moura

Compila una Vez, Arranca Rápido: Bytecode de Python Precompilado en Imágenes de Contenedor

Cómo incluir bytecode precompilado en las imágenes de contenedor de Python hizo que los contenedores nuevos arrancaran unas tres veces más rápido, con las mediciones, los cambios en los Dockerfiles y una forma de comprobar tus propias imágenes.

Publicado: 18 de noviembre de 2026

Artículo original: https://www.cleisson.com/es-ES/blog/python-bytecode-at-build-time

Hace poco ayudé a migrar un grupo de microservicios en Python a uv y Python 3.15. Medí cada servicio antes y después, y un cambio destacó: incluir bytecode precompilado dentro de las imágenes de los contenedores. Con él, los contenedores nuevos arrancan unas tres veces más rápido, y el cambio en sí son unas pocas líneas en cada Dockerfile.

Este artículo explica qué es ese bytecode, por qué nuestros contenedores lo regeneraban en cada arranque y qué cambió cuando dejaron de hacerlo. Los servicios se quedan anónimos aquí; aparecen como Servicio A a F, con sus APIs y workers.

Lo que Python ejecuta de verdad

Cuando Python importa un módulo, no ejecuta el texto del archivo `.py`. Primero compila ese texto a bytecode: una lista de pequeñas instrucciones para la máquina virtual del intérprete. Esta es una función de dos líneas y el bytecode que Python 3.15 genera para ella:

```
def greet(name):
    return "Hello, " + name
```

```
  1           RESUME                   0

  2           LOAD_CONST               0 ('Hello, ')
              LOAD_FAST_BORROW         0 (name)
              BINARY_OP                0 (+)
              RETURN_VALUE
```

Puedes imprimirlo tú mismo con el módulo `dis`. `LOAD_CONST` y `LOAD_FAST_BORROW` ponen la cadena y el argumento en la pila, `BINARY_OP 0 (+)` los suma y `RETURN_VALUE` devuelve el resultado.

Ejecutar estas instrucciones cuesta lo mismo vengan de donde vengan. Lo caro es producirlas: Python lee el texto, lo divide en tokens, construye un árbol sintáctico, averigua a qué se refiere cada nombre y genera y optimiza las instrucciones. Por eso Python guarda el resultado en caché. Después de compilar `greet.py`, escribe `__pycache__/greet.cpython-315.pyc`, y el siguiente import carga ese archivo directamente mientras siga correspondiendo al código fuente.

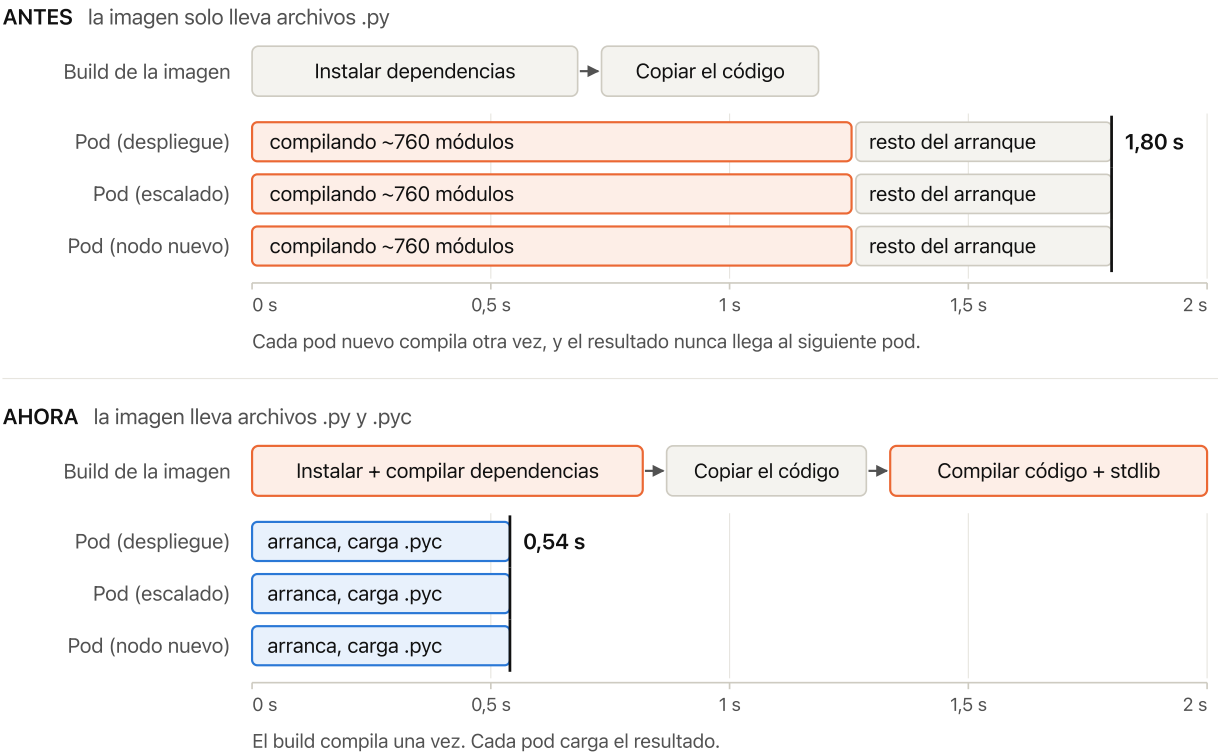
Por qué nuestros contenedores nunca lo conservaban

En nuestras imágenes, esa caché nunca existía cuando arrancaba un contenedor, por tres motivos.

Primero, la imagen base no incluye ninguno. La imagen oficial `python:3.15.2-slim` borra todos los archivos `.pyc` durante su propio build para seguir siendo pequeña. Conté 694 archivos de la biblioteca estándar y otros 404 en su `site-packages`, ninguno con bytecode.

Segundo, los instaladores no compilaban nuestras dependencias. uv, igual que Poetry, instala los paquetes sin bytecode a menos que se lo pidas, y deja la compilación para el primer import. pip es la excepción: compila por defecto.

Tercero, los contenedores no podían conservar lo que compilaban. Algunas cargas de trabajo se ejecutan con un usuario sin privilegios de root sobre un sistema de archivos raíz de solo lectura, y algunas además definen `PYTHONDONTWRITEBYTECODE=1`. Compilan en memoria y descartan el resultado cuando el proceso termina, así que cada arranque vuelve a pagar el coste, igual que cada ejecución de cron y cada health probe que arranca un proceso de Python. Las cargas de trabajo que se ejecutan como root sobre un sistema de archivos con escritura sí escribían archivos `.pyc`, pero en la capa del propio contenedor, que desaparece con él. Cada despliegue, cada escalado y cada reinicio en un nodo nuevo empezaba desde cero.



La figura está a escala y usa una de las APIs, la del Servicio F, ejecutándose con 1 CPU: 1,80 segundos desde que arranca el contenedor hasta que está listo sin bytecode, y 0,54 segundos con él. La diferencia de 1,26 segundos es compilación, y antes del cambio la pagaba cada contenedor nuevo.

La solución: compilar una vez, durante el build

Ahora son los Dockerfiles los que compilan, en este orden:

```
# Dependencias: uv compila todo lo que instala.
RUN UV_COMPILE_BYTECODE=1 \
      uv sync --locked --no-default-groups --no-install-project

# Primero copia todo el código Python...
COPY app /app/app
COPY gunicorn.conf.py entrypoint.sh /app/
```

```
# ...y después compila la aplicación y la biblioteca estándar.
RUN python -m compileall -q -j 0 /app \
      "$(python -c 'import sysconfig; print(sysconfig.get_paths()["stdlib"])')"
```

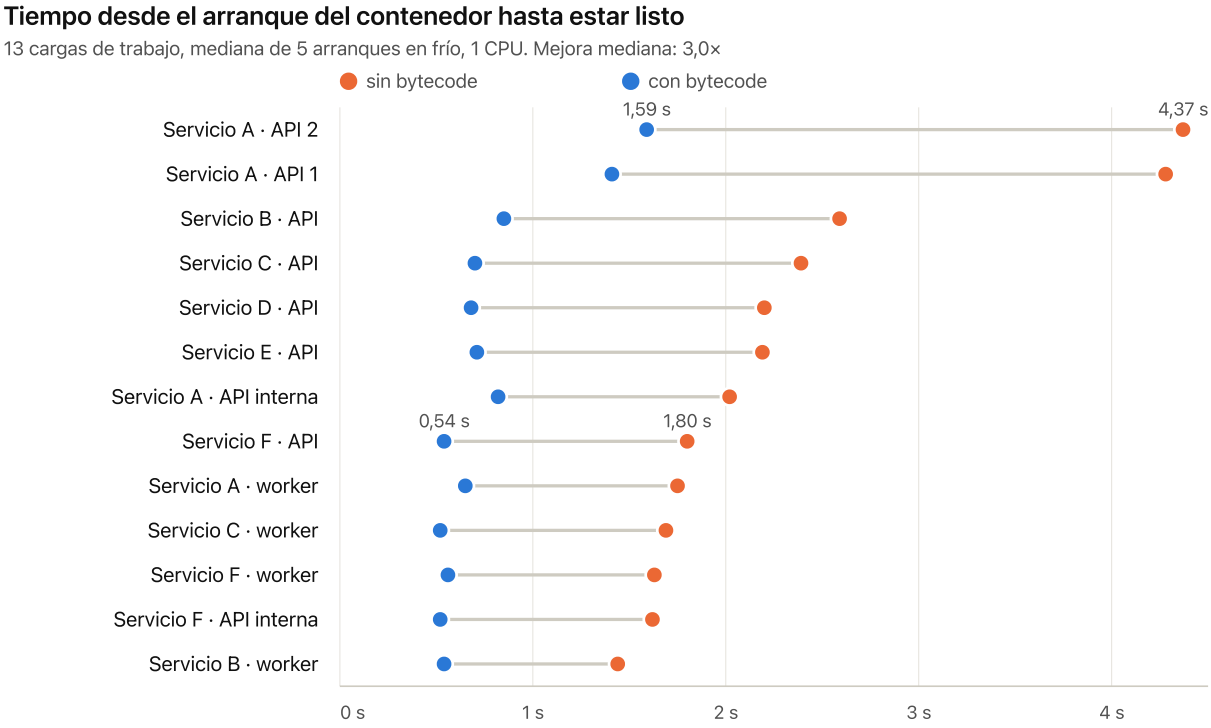
`UV_COMPILE_BYTECODE=1` hace que uv compile cada paquete que instala en el entorno virtual. Viene desactivado por defecto, y la guía de Docker de uv lo recomienda para imágenes.

El paso de `compileall` cubre lo que uv no instala: nuestro propio código y la biblioteca estándar. `-q` solo muestra los errores y `-j 0` usa todas las CPUs. Leo la ruta de la biblioteca estándar del propio intérprete porque `compileall` termina con éxito con un directorio que no existe, así que una ruta mal escrita se saltaría sin que fallara el build.

El orden importa. Todo lo que se copie después del paso de compilación llega a la imagen sin bytecode, así que el paso va después del último `COPY` de código Python. En un build multi-stage, va en la etapa que se convierte en la imagen final. Uno de los servicios genera siete imágenes a partir de un solo Dockerfile. Ahí, la biblioteca estándar, las dependencias y el código compartido se compilan en capas que reutilizan las siete imágenes, y cada imagen compila solo su propio paquete, al final.

Qué cambió

En la mayoría de los servicios medí tres versiones: la que se ejecutaba antes de la migración, la migración a uv y Python 3.15 sin bytecode, y la misma migración con el paso de bytecode. Cada carga de trabajo se ejecutó igual que en producción, con su entrypoint, su usuario y su límite de memoria reales, en Docker en un portátil, con imágenes linux/arm64 y 1 CPU. La migración por sí sola dejó el arranque más o menos donde estaba. Añadir bytecode lo redujo en unos dos tercios.



El gráfico muestra las 13 cargas de trabajo en las que el bytecode era el único cambio entre las dos mediciones que podía afectar al arranque. Cada punto es la mediana de cinco arranques en frío. La mejora mediana es de 3,0×, con un rango de 2,5× a 3,4×.

Otros resultados:

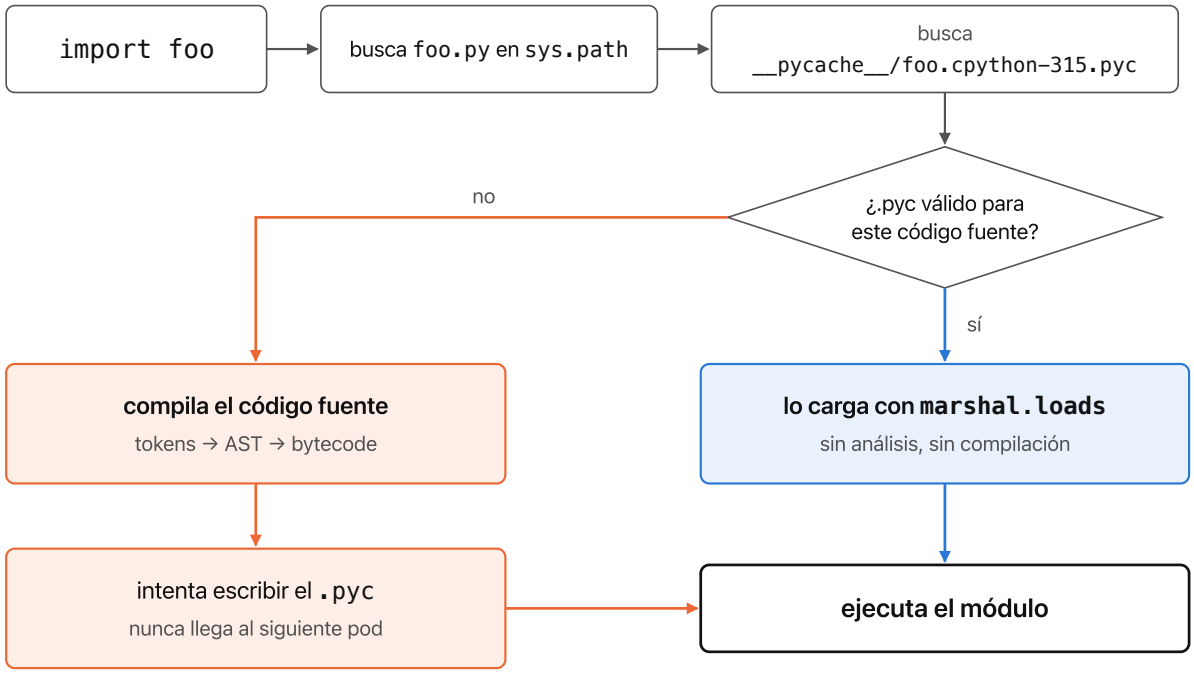
- Los módulos compilados en el arranque bajaron a 0 en todas las cargas de trabajo que medí. Antes, la mayoría compilaba varios cientos de módulos en cada arranque, hasta unos 1100.
- La memoria también bajó un poco: de 9 a 17 MiB menos por proceso en uno de los servicios, solo por el bytecode.
- En los entornos de prueba desplegados, los contenedores nuevos tenían bytecode válido para el 100 % de sus archivos Python. Los contenedores a los que sustituyeron tenían entre el 0 % y el 83 %.

El coste es pequeño. Los archivos de bytecode ocupan de 3 a 16 MB por imagen comprimida, y el build tarda unos segundos más; en uno de los servicios, el build en frío pasó de 45,0 a 49,7 segundos. Aun así, todas las imágenes acabaron siendo más pequeñas que antes de la migración.

Cómo funciona por dentro

Qué hace `import`

CPython toma la misma decisión con cada módulo que importa:



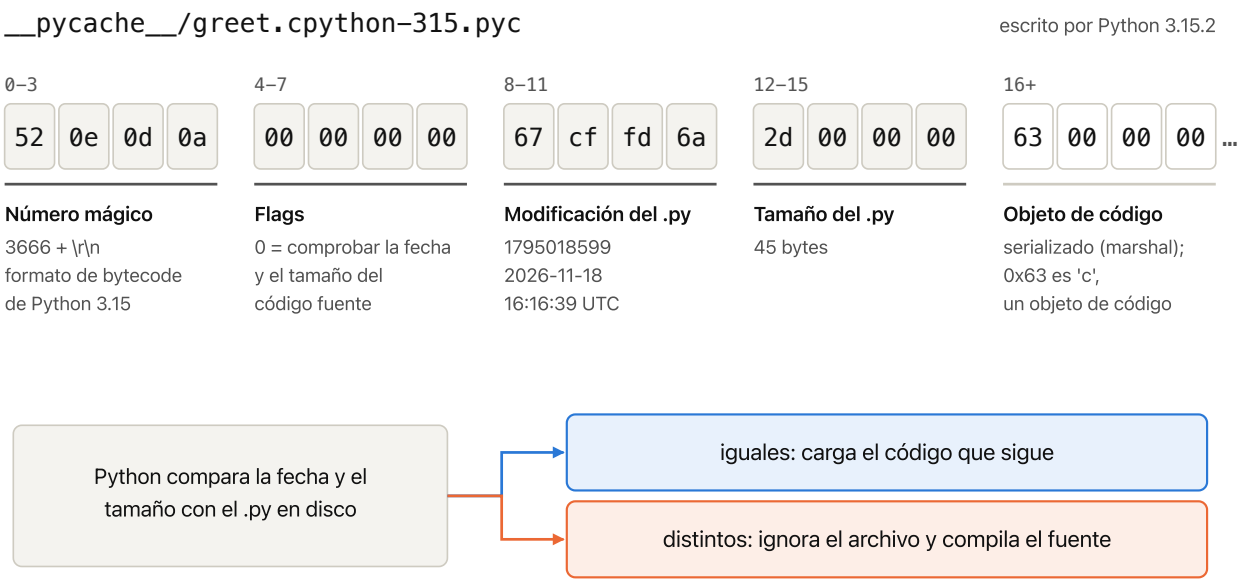
Los dos caminos terminan en el mismo objeto de código.

1. Encuentra `foo.py` en `sys.path`.
2. Calcula la ruta de la caché, `__pycache__/foo.cpython-315.pyc`. La etiqueta `cpython-315` mantiene separados los archivos de cada versión de Python.
3. Si ese archivo existe, lee su cabecera y la compara con el código fuente. Si coinciden, `marshal.loads` reconstruye el objeto de código directamente desde el archivo.
4. Si no, compila el código fuente y después intenta escribir el `.pyc`. Con `PYTHONDONTWRITEBYTECODE=1` se salta la escritura. En un sistema de archivos de solo lectura, la escritura falla y Python ignora el error.
5. Ejecuta el código del módulo, que es el mismo en los dos caminos.

Antes del cambio, todos los módulos de nuestros contenedores tomaban el camino de la compilación en cada arranque. Ahora todos toman el de la carga.

Los 16 bytes que deciden

Todo archivo `.pyc` empieza con una cabecera de 16 bytes, y Python solo lee esa cabecera para decidir si se fía del archivo. Estos son los bytes reales del ejemplo `greet`:

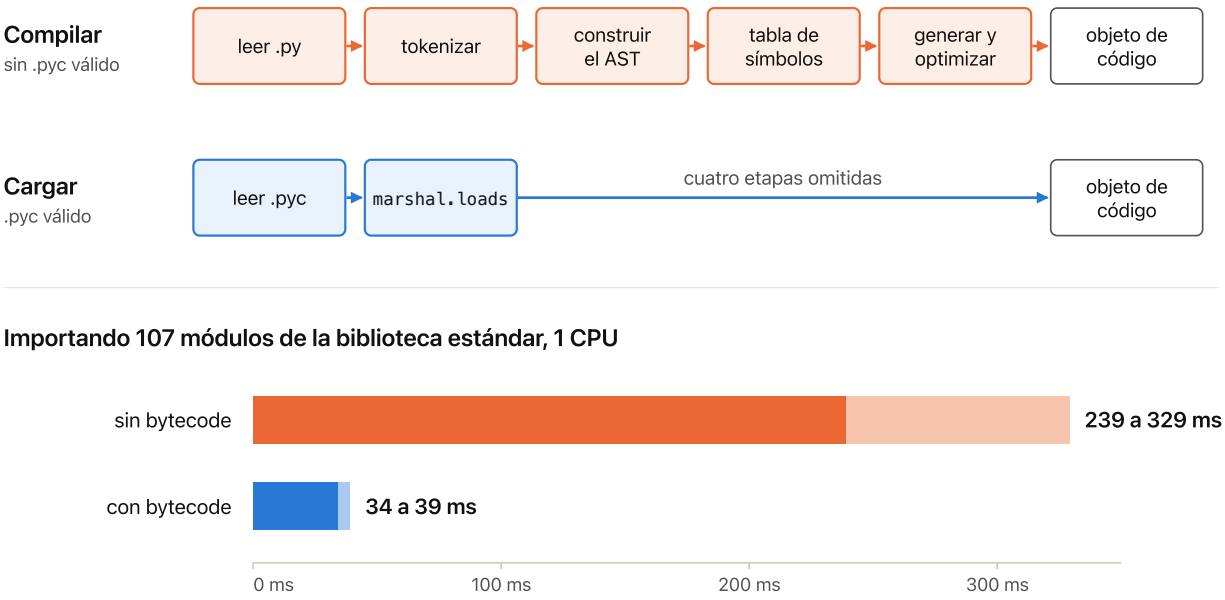


La imagen no cambia después del build, así que en el contenedor la comprobación siempre pasa.

El número mágico identifica el formato del bytecode, que cambia entre versiones de Python. Los flags indican cómo validar el archivo, y 0 significa "comparar la fecha y el tamaño". Después vienen la fecha de modificación y el tamaño del archivo fuente, guardados con el byte menos significativo primero. Si ambos coinciden con el `.py` en disco, Python carga el código que sigue. Si no, ignora el archivo y compila el código fuente.

Una imagen no cambia después del build, así que dentro de un contenedor la comprobación siempre pasa. Otra versión de Python tiene otro número mágico y otro nombre de archivo, así que nunca aprovecha el bytecode de una versión distinta. Los builds que definen `SOURCE_DATE_EPOCH` generan archivos `.pyc` basados en hash (PEP 552), que guardan un hash del código fuente en lugar de la fecha y el tamaño.

Por qué compilar cuesta más que cargar



Pico de memoria: 29,8 MiB sin bytecode, 24,2 MiB con él.

Cuatro ejecuciones, cada una la mediana de siete arranques, en la imagen python:3.15.2-slim original.

Los dos caminos terminan en el mismo objeto de código. Cargar se salta las etapas intermedias, porque `marshal.loads` solo reconstruye objetos a partir de bytes que ya están organizados.

Para ver la diferencia sin nada de nuestro código, medí la imagen original `python:3.15.2-slim` con 1 CPU mientras importaba siete paquetes habituales de la biblioteca estándar, 107 módulos en total. Tardó de 239 a 329 ms sin bytecode y de 34 a 39 ms con él, en cuatro ejecuciones que tomaban cada una la mediana de siete arranques. El pico de memoria fue de 29,8 MiB sin bytecode y de 24,2 MiB con él.

La memoria baja porque compilar crea muchos objetos temporales: tokens, nodos del árbol sintáctico y tablas de símbolos. Cuando se liberan, el proceso suele conservar esa memoria para reutilizarla en lugar de devolverla al sistema operativo, así que un proceso que compiló al arrancar sigue siendo más grande durante toda su vida.

Pruébalo tú mismo

Estos comandos repiten el experimento con Docker:

```
IMAGE=public.ecr.aws/docker/library/python:3.15.2-slim
TIMED='import time; t = time.perf_counter(); import asyncio, email.message,
http.client, json, logging, ssl, urllib.request; print(round((time.perf_counter()
- t) * 1000), "ms")'

# Sin bytecode: la imagen original, de solo lectura.
```

```
docker run --rm --read-only --cpus 1 -e TIMED="$TIMED" "$IMAGE" \
  sh -c 'for i in 1 2 3; do python -c "$TIMED"; done'

# Con bytecode: primero compila la biblioteca estándar y después mide los mismos
imports.
docker run --rm --cpus 1 -e TIMED="$TIMED" "$IMAGE" \
  sh -c 'python -m compileall -q -j 0 /usr/local/lib/python3.15 && for i in 1 2
3; do python -c "$TIMED"; done'
```

Cada comando imprime tres tiempos. Una sola ejecución varía, así que compara varias.

Para comprobar una imagen tuya, arranca Python con `-v` e importa tu paquete. Python registra cada objeto de código que construye: una ruta que termina en `.py` significa que compiló ese módulo, y una ruta `.pyc` entre comillas significa que cargó bytecode. Este comando cuenta los compilados:

```
docker run --rm --read-only <image> python -B -v -c "import <package>" 2>&1 \
  | grep -c '^# code object from /.*\.py$'
```

Debería imprimir 0. La imagen base original imprime 20 con `import json`.

Para un contenedor en ejecución, usé un script pequeño que comprueba, para cada archivo `.py`, si hay un `.pyc` con una cabecera que Python aceptaría. No importa nada de la aplicación y, ejecutado con `-B`, no escribe nada, así que es seguro en contenedores de solo lectura:

```
import importlib.util, json, os, sys, sysconfig

def check(root):
    total = valid = 0
    for directory, subdirs, files in os.walk(root):
        subdirs[:] = [d for d in subdirs if d != "__pycache__"]
        for name in files:
            if not name.endswith(".py"):
                continue
            source = os.path.join(directory, name)
            total += 1
            try:
                info = os.stat(source)
                with open(importlib.util.cache_from_source(source), "rb") as
handle:
                    header = handle.read(16)
            except OSError:
```

```
                continue
            expected = (importlib.util.MAGIC_NUMBER + bytes(4)
                        + (int(info.st_mtime) & 0xFFFFFFFF).to_bytes(4, "little")
                        + (info.st_size & 0xFFFFFFFF).to_bytes(4, "little"))
            valid += header == expected
    return {"sources": total, "valid_bytecode": valid}

roots = [*sys.argv[1:], sysconfig.get_path("purelib"),
sysconfig.get_path("stdlib")]
print(json.dumps({root: check(root) for root in roots if os.path.isdir(root)},
indent=2))
```

Guárdalo como `check_bytecode.py` y pásalo al contenedor por la entrada estándar, por ejemplo con `kubectl exec -i <pod> -- python -B - /app < check_bytecode.py`. Cada directorio debería indicar tantos archivos válidos como archivos fuente. En la imagen base original, ninguno de los 1098 archivos de su directorio de bibliotecas tiene bytecode válido.

Posibles trampas

- Conserva los archivos `.py`. Python puede ejecutarse solo con los `.pyc`, y quitar las fuentes ahorra espacio, pero hay código que encuentra sus módulos buscando archivos `.py`. Un registro de plugins que busque archivos llamados `processor.py`, por ejemplo, no encontraría nada.
- Compila todos los directorios desde los que se importa tu código, incluido el código compartido instalado desde otra ruta. Un directorio olvidado sigue funcionando; simplemente sigue compilando en el arranque.
- `PYTHONDONTWRITEBYTECODE=1` puede quedarse, igual que el sistema de archivos de solo lectura. Solo impiden la escritura; la lectura funciona igual.
- Ejecutar con `-O` o con `-OO` hace que Python busque archivos `.opt-1.pyc` o `.opt-2.pyc`. Sin ellos, compila como antes: más lento, pero nunca mal.
- El bytecode es el mismo que Python generaría al importar, solo que creado antes por el mismo intérprete. Si algún `.pyc` no coincidiera con su código fuente, Python lo ignoraría y compilaría, así que el peor caso es el tiempo de arranque de antes.

Si construyes imágenes de Python, comprueba si incluyen bytecode. Las nuestras no lo incluían, y arreglarlo costó unas pocas líneas y unos pocos megabytes por imagen.