Cleisson de Oliveira Moura

Compile uma Vez, Inicie Rápido: Bytecode Python Pré-Compilado em Imagens de Contêiner

Como incluir bytecode pré-compilado nas imagens de contêiner Python fez os novos contêineres iniciarem cerca de três vezes mais rápido, com as medições, as mudanças nos Dockerfiles e uma forma de verificar as suas próprias imagens.

Publicado: 18 de novembro de 2026

Artigo original: https://www.cleisson.com/pt-BR/blog/python-bytecode-at-build-time

Recentemente, ajudei a migrar um grupo de microsserviços Python para o uv e o Python 3.15. Medi cada serviço antes e depois, e uma mudança se destacou: incluir bytecode pré-compilado dentro das imagens dos contêineres. Com ela, os novos contêineres iniciam cerca de três vezes mais rápido, e a mudança em si são poucas linhas em cada Dockerfile.

Este artigo explica o que é esse bytecode, por que nossos contêineres o recriavam a cada início e o que mudou quando deixaram de fazer isso. Os serviços ficam anônimos aqui; aparecem como Serviço A a F, com suas APIs e workers.

O que o Python realmente executa

Quando o Python importa um módulo, ele não executa o texto do arquivo `.py`. Primeiro, compila esse texto em bytecode: uma lista de pequenas instruções para a máquina virtual do interpretador. Esta é uma função de duas linhas e o bytecode que o Python 3.15 gera para ela:

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

```
  1           RESUME                   0

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

Você pode imprimir isso com o módulo `dis`. `LOAD_CONST` e `LOAD_FAST_BORROW` colocam a string e o argumento na pilha, `BINARY_OP 0 (+)` soma os dois e `RETURN_VALUE` devolve o resultado.

Executar essas instruções custa o mesmo, venham de onde vierem. A parte cara é produzi-las: o Python lê o texto, separa em tokens, monta uma árvore sintática, descobre a que cada nome se refere e gera e otimiza as instruções. Por isso, o Python guarda o resultado em cache. Depois de compilar `greet.py`, ele grava `__pycache__/greet.cpython-315.pyc`, e o próximo import carrega esse arquivo direto, desde que ele ainda corresponda ao código-fonte.

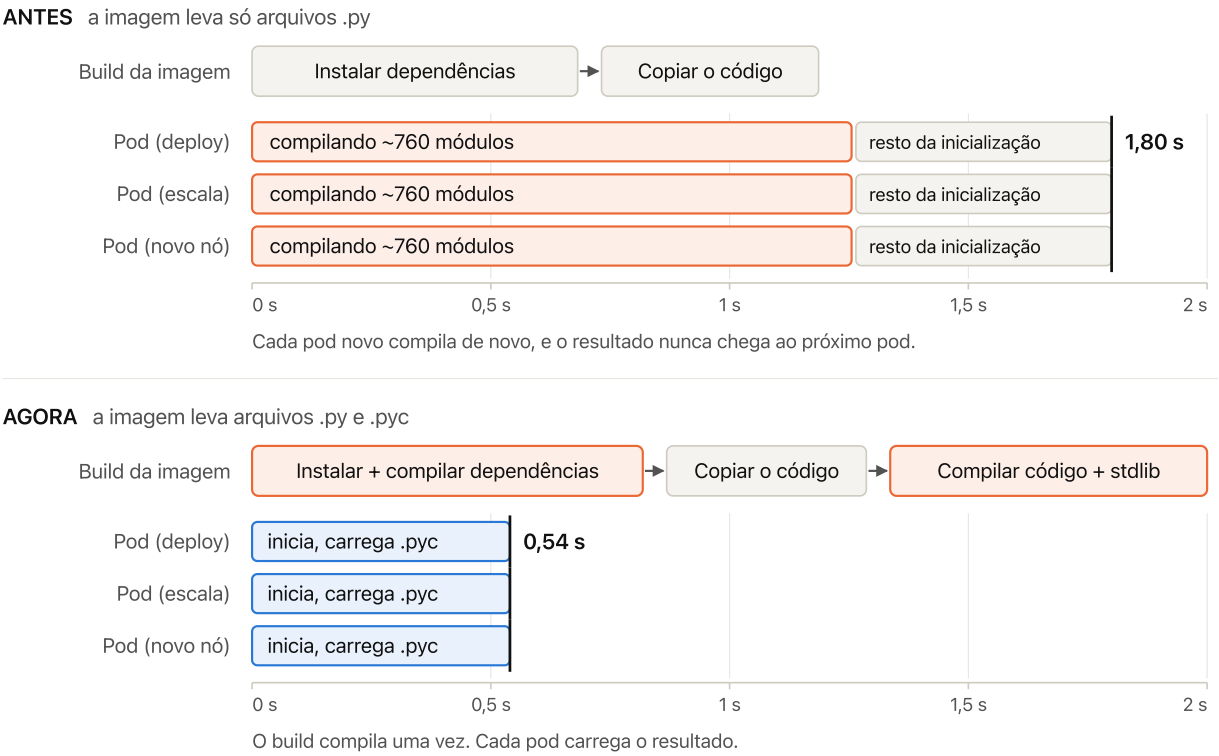
Por que nossos contêineres nunca guardavam esse cache

Nas nossas imagens, esse cache nunca existia quando um contêiner iniciava, por três motivos.

Primeiro, a imagem base não traz nenhum. A imagem oficial `python:3.15.2-slim` apaga todos os arquivos `.pyc` durante o próprio build para continuar pequena. Contei 694 arquivos da biblioteca padrão e mais 404 no `site-packages`, nenhum deles com bytecode.

Segundo, os instaladores não compilavam nossas dependências. O uv, assim como o Poetry, instala pacotes sem bytecode, a menos que você peça, e deixa a compilação para o primeiro import. O pip é a exceção: ele compila por padrão.

Terceiro, os contêineres não conseguiam guardar o que compilavam. Alguns workloads rodam com um usuário sem privilégios de root, em um sistema de arquivos raiz somente leitura, e alguns também definem `PYTHONDONTWRITEBYTECODE=1`. Eles compilam em memória e descartam o resultado quando o processo termina, então cada início paga de novo, assim como cada execução de cron e cada health probe que inicia um processo Python. Os workloads que rodam como root em um sistema de arquivos gravável chegavam a gravar arquivos `.pyc`, mas na camada do próprio contêiner, que desaparece junto com ele. Cada deploy, cada aumento de réplicas e cada reinício em um nó novo começava do zero.



A figura está em escala e usa uma das APIs, a do Serviço F, rodando com 1 CPU: 1,80 s do início do contêiner até ficar pronta sem bytecode, e 0,54 s com ele. A diferença de 1,26 s é compilação, e antes da mudança todo contêiner novo pagava por ela.

A correção: compilar uma vez, durante o build

Agora quem compila são os Dockerfiles, nesta ordem:

```
# Dependências: o uv compila tudo o que instala.
RUN UV_COMPILE_BYTECODE=1 \
      uv sync --locked --no-default-groups --no-install-project

# Primeiro, copie todo o código Python...
COPY app /app/app
COPY gunicorn.conf.py entrypoint.sh /app/
```

```
# ...depois compile a aplicação e a biblioteca padrão.
RUN python -m compileall -q -j 0 /app \
      "$(python -c 'import sysconfig; print(sysconfig.get_paths()["stdlib"])')"
```

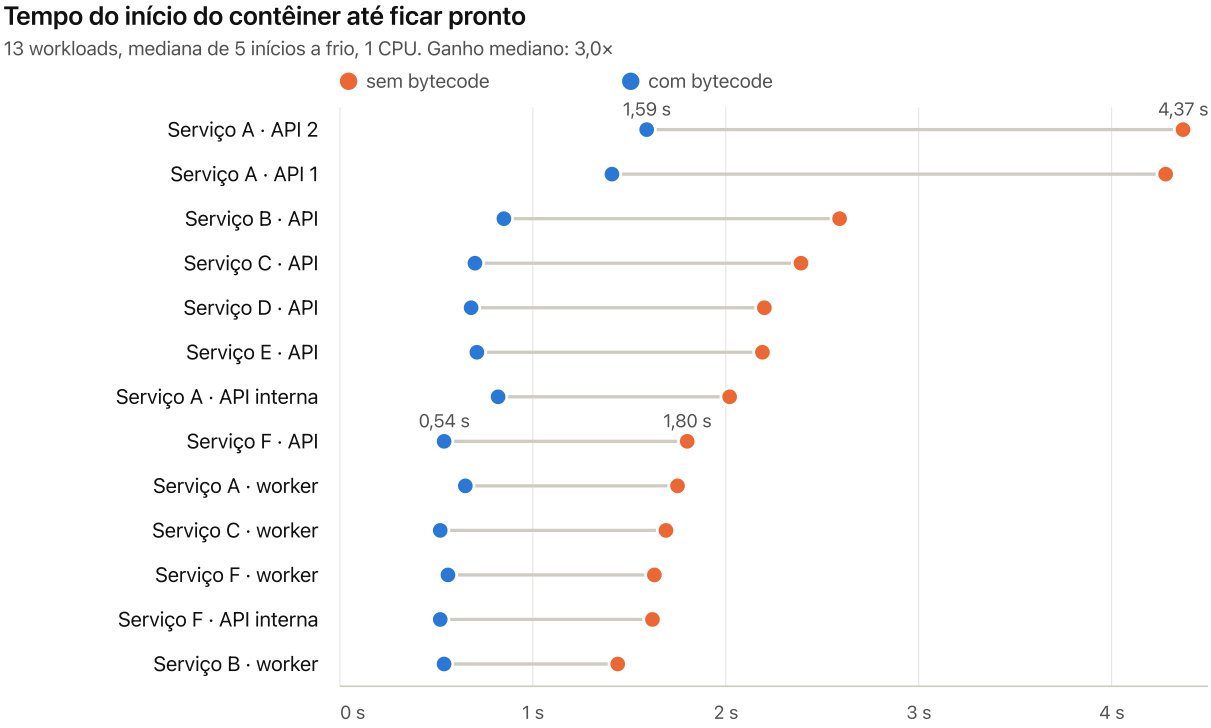
`UV_COMPILE_BYTECODE=1` faz o uv compilar cada pacote que instala no ambiente virtual. Vem desligado por padrão, e o guia de Docker do uv recomenda ligá-lo em imagens.

O passo do `compileall` cobre o que o uv não instala: o nosso próprio código e a biblioteca padrão. `-q` mostra só os erros e `-j 0` usa todas as CPUs. Leio o caminho da biblioteca padrão do próprio interpretador porque o `compileall` termina com sucesso para um diretório que não existe, então um caminho digitado errado seria pulado sem quebrar o build.

A ordem importa. Tudo o que for copiado depois do passo de compilação vai para a imagem sem bytecode, então o passo vem depois do último `COPY` de código Python. Em um build multi-stage, ele fica no estágio que vira a imagem final. Um dos serviços gera sete imagens a partir de um único Dockerfile. Nele, a biblioteca padrão, as dependências e o código compartilhado são compilados em camadas que as sete imagens reaproveitam, e cada imagem compila só o seu próprio pacote, por último.

O que mudou

Na maioria dos serviços, medi três versões: a que rodava antes da migração, a migração para o uv e o Python 3.15 sem bytecode e a mesma migração com o passo de bytecode. Cada workload rodou do jeito que roda em produção, com o entrypoint, o usuário e o limite de memória reais, no Docker de um notebook, com imagens linux/arm64 e 1 CPU. A migração sozinha deixou a inicialização mais ou menos onde estava. Com o bytecode, ela caiu cerca de dois terços.



O gráfico mostra os 13 workloads em que o bytecode era a única mudança entre as duas medições capaz de afetar a inicialização. Cada ponto é a mediana de cinco inícios a frio. O ganho mediano é de 3,0×, e vai de 2,5× a 3,4×.

Alguns outros resultados:

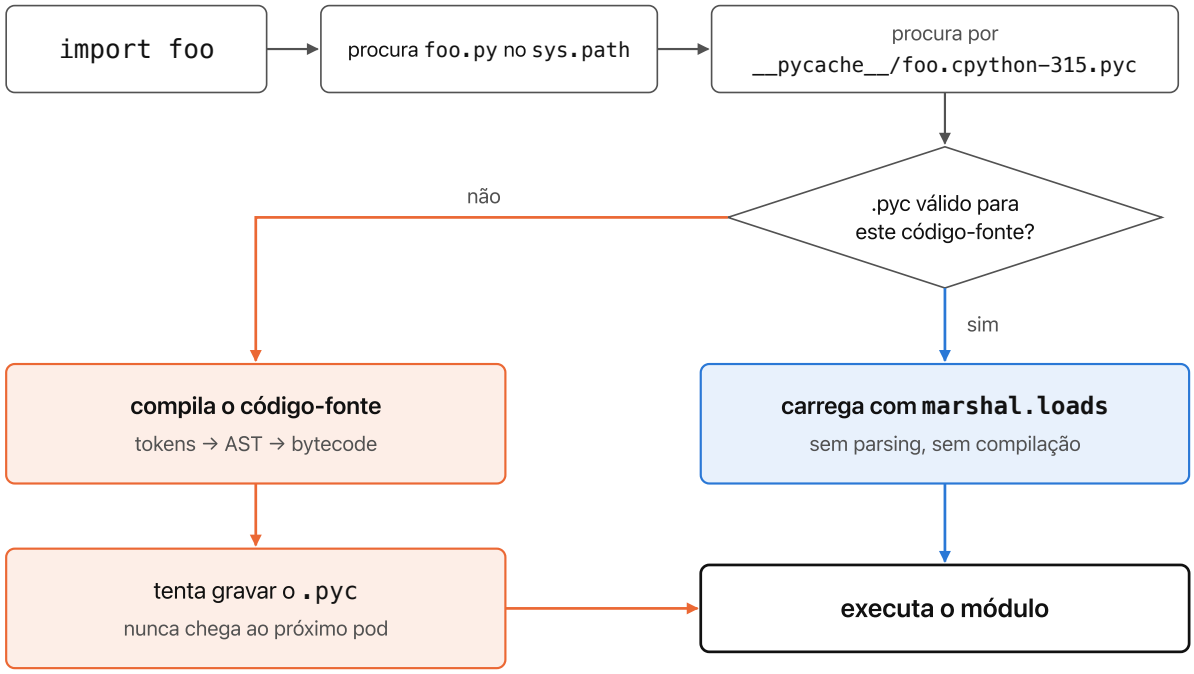
- Os módulos compilados na inicialização caíram para 0 em todos os workloads que medi. Antes, a maioria compilava várias centenas de módulos a cada início, chegando a cerca de 1.100.
- A memória também caiu um pouco: de 9 a 17 MiB a menos por processo em um dos serviços, só por causa do bytecode.
- Nos ambientes de teste já implantados, os novos contêineres tinham bytecode válido para 100% dos seus arquivos Python. Os contêineres que eles substituíram tinham entre 0% e 83%.

O custo é pequeno. Os arquivos de bytecode ocupam de 3 a 16 MB por imagem comprimida, e o build leva alguns segundos a mais; em um dos serviços, o build a frio passou de 45,0 para 49,7 segundos. Mesmo assim, todas as imagens ficaram menores do que eram antes da migração.

Por baixo dos panos

O que o `import` faz

O CPython toma a mesma decisão para cada módulo que importa:



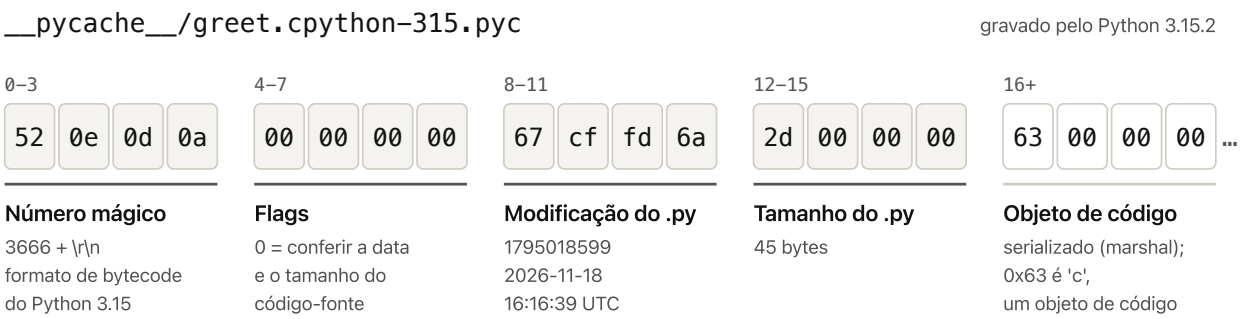
Os dois caminhos terminam no mesmo objeto de código.

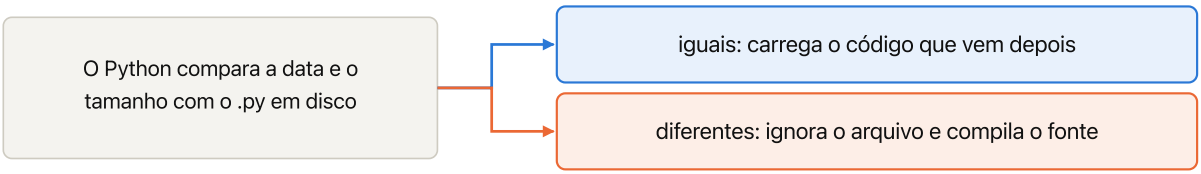
1. Encontra `foo.py` no `sys.path`.
2. Calcula o caminho do cache, `__pycache__/foo.cpython-315.pyc`. A tag `cpython-315` mantém separados os arquivos de cada versão do Python.
3. Se esse arquivo existe, lê o cabeçalho e o compara com o código-fonte. Se baterem, `marshal.loads` reconstrói o objeto de código direto do arquivo.
4. Caso contrário, compila o código-fonte e depois tenta gravar o `.pyc`. Com `PYTHONDONTWRITEBYTECODE=1`, ele pula a gravação. Em um sistema de arquivos somente leitura, a gravação falha e o Python ignora o erro.
5. Executa o código do módulo, que é o mesmo nos dois caminhos.

Antes da mudança, todo módulo dos nossos contêineres seguia o caminho da compilação a cada início. Agora, todos seguem o caminho do carregamento.

Os 16 bytes que decidem

Todo arquivo `.pyc` começa com um cabeçalho de 16 bytes, e o Python lê só esse cabeçalho para decidir se confia no arquivo. Estes são os bytes reais do exemplo `greet`:



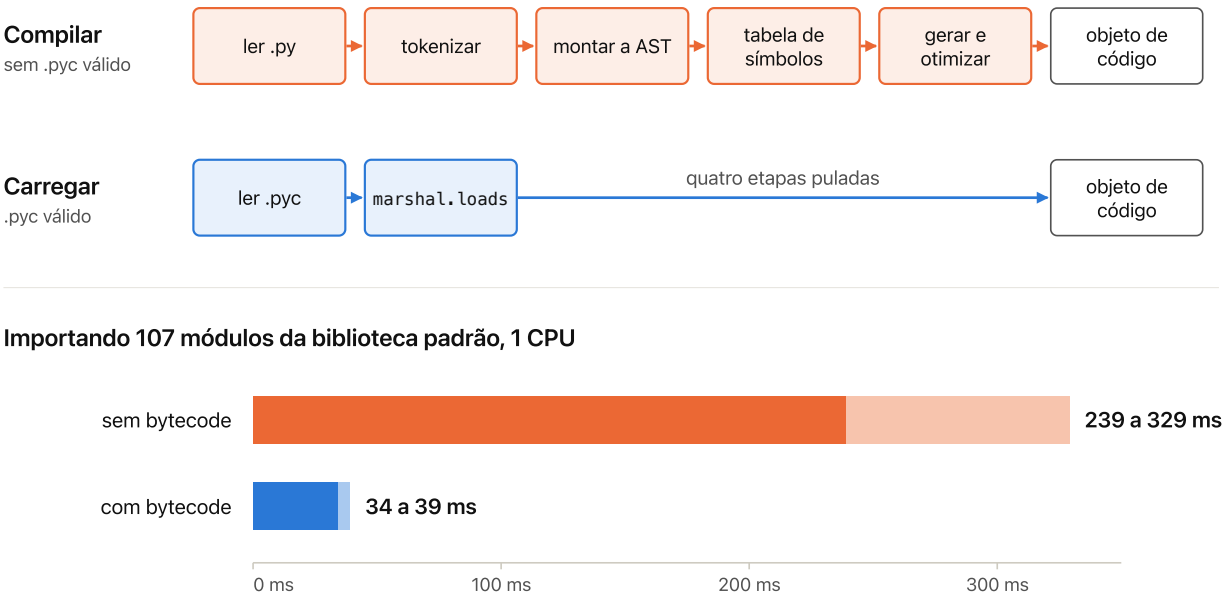


A imagem não muda depois do build, então no contêiner a checagem sempre passa.

O número mágico identifica o formato do bytecode, que muda entre versões do Python. As flags dizem como validar o arquivo, e 0 significa "comparar data e tamanho". Depois vêm a data de modificação e o tamanho do arquivo-fonte, gravados com o byte menos significativo primeiro. Se os dois baterem com o `.py` em disco, o Python carrega o código que vem a seguir. Se não, ignora o arquivo e compila o código-fonte.

A imagem não muda depois do build, então, dentro do contêiner, a checagem sempre passa. Outra versão do Python tem outro número mágico e outro nome de arquivo, então nunca aproveita o bytecode de uma versão diferente. Builds que definem `SOURCE_DATE_EPOCH` geram arquivos `.pyc` baseados em hash (PEP 552), que guardam um hash do código-fonte no lugar da data e do tamanho.

Por que compilar custa mais do que carregar



Pico de memória: 29,8 MiB sem bytecode, 24,2 MiB com ele.

Quatro execuções, cada uma a mediana de sete inícios, na imagem python:3.15.2-slim original.

Os dois caminhos terminam no mesmo objeto de código. Carregar pula as etapas do meio, porque `marshal.loads` só reconstrói objetos a partir de bytes que já estão organizados.

Para ver a diferença sem nenhum código nosso, medi a imagem original `python:3.15.2-slim` com 1 CPU importando sete pacotes comuns da biblioteca padrão, 107 módulos no total. Levou de 239 a 329 ms sem bytecode e de 34 a 39 ms com ele, em quatro execuções, cada uma usando a mediana de sete inícios. O pico de memória foi de 29,8 MiB sem bytecode e 24,2 MiB com ele.

A memória cai porque compilar cria muitos objetos temporários: tokens, nós da árvore sintática e tabelas de símbolos. Quando eles são liberados, o processo em geral guarda essa memória para reutilizar em vez de devolvê-la ao sistema operacional, então um processo que compilou ao iniciar continua maior durante toda a sua vida.

Teste você mesmo

Estes comandos repetem o experimento com 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")'

# Sem bytecode: a imagem original, somente leitura.
docker run --rm --read-only --cpus 1 -e TIMED="$TIMED" "$IMAGE" \
```

```
  sh -c 'for i in 1 2 3; do python -c "$TIMED"; done'

# Com bytecode: compila primeiro a biblioteca padrão e depois mede os mesmos
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 três tempos. Execuções isoladas variam, então compare algumas.

Para conferir uma imagem sua, inicie o Python com `-v` e importe o seu pacote. O Python registra cada objeto de código que monta: um caminho terminado em `.py` significa que ele compilou aquele módulo, e um caminho `.pyc` entre aspas significa que carregou bytecode. Este comando conta os compilados:

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

Ele deve imprimir 0. A imagem base original imprime 20 para `import json`.

Para um contêiner em execução, usei um script pequeno que verifica, para cada arquivo `.py`, se existe um `.pyc` com um cabeçalho que o Python aceitaria. Ele não importa nada da aplicação e, rodando com `-B`, não grava nada, então é seguro em contêineres somente leitura:

```
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))
```

Salve como `check_bytecode.py` e envie para o contêiner pela entrada padrão, por exemplo com `kubectl exec -i <pod> -- python -B - /app < check_bytecode.py`. Cada diretório deve informar tantos arquivos válidos quanto arquivos-fonte. Na imagem base original, nenhum dos 1.098 arquivos do diretório de bibliotecas tem bytecode válido.

Armadilhas

- Mantenha os arquivos `.py`. O Python consegue rodar só com os `.pyc`, e remover os fontes economiza espaço, mas há código que encontra seus módulos procurando arquivos `.py`. Um registro de plugins que procura arquivos chamados `processor.py`, por exemplo, não encontraria nada.
- Compile todos os diretórios de onde o seu código é importado, incluindo código compartilhado instalado a partir de outro caminho. Um diretório esquecido continua funcionando; só continua compilando na inicialização.
- `PYTHONDONTWRITEBYTECODE=1` pode continuar, assim como o sistema de arquivos somente leitura. Eles só impedem a gravação; a leitura funciona igual.
- Rodar com `-O` ou `-OO` faz o Python procurar arquivos `.opt-1.pyc` ou `.opt-2.pyc`. Sem eles, compila como antes: mais devagar, mas nunca errado.
- O bytecode é o mesmo que o Python geraria no momento do import, só que produzido antes, pelo mesmo interpretador. Se algum `.pyc` não batesse com o seu código-fonte, o Python o ignoraria e compilaria, então o pior caso é o tempo de inicialização antigo.

Se você cria imagens Python, confira se elas levam bytecode. As nossas não levavam, e corrigir isso custou algumas linhas e alguns megabytes por imagem.