Containers para Aplicações

Tópicos de Informática para Automação

Mário Antunes

October 20, 2025

Exercícios

Exercícios Práticos: Flatpak & AppImage

Objetivo: Esta aula irá guiá-lo através dos fundamentos do empacotamento de aplicações. Começará com um simples "Hello World" e progredirá até ao empacotamento de uma aplicação gráfica (GUI) completa em Python, com as suas dependências.

0. Setup: Configurar o Ambiente de Trabalho

Primeiro, temos de instalar todas as ferramentas necessárias para construir e testar os nossos pacotes de aplicação.

1. **Atualizar o sistema:** Este comando descarrega a lista mais recente de software disponível e atualiza todos os pacotes atualmente instalados para as suas versões mais recentes.

```
$ sudo apt update && sudo apt full-upgrade -y
```

2. **Instalar ferramentas:** Este comando instala todos os componentes necessários para a nossa aula.

- `curl` & `wget`: Utilitários para descarregar ficheiros da internet.
- `file`: Um utilitário para identificar tipos de ficheiro.
- `libfuse2`: Uma biblioteca necessária ao AppImage para "montar" (mount) o pacote da aplicação como um sistema de ficheiros virtual (virtual filesystem).
- `flatpak`: A ferramenta de linha de comandos para executar e gerir aplicações Flatpak.
- `flatpak-builder`: A ferramenta específica usada para construir pacotes Flatpak a partir de um ficheiro `manifest`.
- `python3`, `python3-pip`, `python3-venv`: O interpretador Python, o seu gestor de pacotes (`pip`), e a ferramenta de **virtual environment** (`venv`).

```
$ sudo apt install curl wget file libfuse2 flatpak \
flatpak-builder python3 python3-pip python3-venv
```

3. **Adicionar o Flathub:** Este comando adiciona o repositório **Flathub** à configuração do Flatpak do seu sistema, mas apenas para o seu utilizador local (`--user`). Um "repositório" (ou "remote") é um servidor que aloja aplicações e `runtimes` do Flatpak. O Flathub é o maior e mais comum repositório, e precisamos dele para descarregar os "SDKs" (Software Development Kits) necessários para a compilação.

```
$ flatpak --user remote-add --if-not-exists \
flathub https://flathub.org/repo/flathub.flatpakrepo
```

4. **Instalar o `appimagetool`:** Isto descarrega o programa `appimagetool`, que é o que comprime um diretório `AppDir` num único ficheiro executável AppImage. Tornamo-lo executável (`chmod +x`) e movemo-lo para `~/.local/bin`, um diretório padrão para programas instalados pelo utilizador.

```
$ mkdir -p ~/.local/bin
$ wget -O appimagetool \
"https://github.com/AppImage/AppImageKit/\
releases/download/continuous/appimagetool-x86_64.AppImage"
```

```
$ chmod +x appimagetool
$ mv appimagetool ~/.local/bin/
```

5. **Aplicar a alteração ao PATH:** O PATH é uma variável de ambiente que indica à sua `shell` (como o `bash`) em que diretórios procurar por programas executáveis. Por defeito, `~/.local/bin` nem sempre está no PATH. Editamos o `~/.bashrc` (um ficheiro que é executado sempre que abre um novo terminal) para adicionar este diretório ao seu PATH. Isto torna o `appimagetool` executável a partir de qualquer lugar.

   Pode usar o `nano` para editar o ficheiro: `nano ~/.bashrc`. Adicione a seguinte configuração à *última* linha do ficheiro:

```
export PATH=${HOME}/.local/bin${PATH:+:${PATH}}
```

6. **Termine a sessão e inicie novamente (Log out e log back in).** Isto recarrega o seu ficheiro `~/.bashrc` e aplica a alteração ao PATH. Para verificar que está a funcionar, abra um novo terminal e escreva o seguinte comando. Deverá ver a informação da versão da ferramenta.

```
$ appimagetool --version
```

---

1. "Hello World" 🌍

Vamos empacotar um `script` de `shell` simples.

1.A: Flatpak "Hello World"

O Flatpak usa um ficheiro "manifest" (em formato YAML) para definir tudo sobre a aplicação e como a construir.

1. Crie um diretório para este exercício:

```
$ mkdir ex1-flatpak && cd ex1-flatpak
```

2. Crie o `script` da aplicação, chamado `hello.sh`. Este pode ser criado com qualquer editor; uma possibilidade é usar o `nano`: `nano hello.sh`

```
#!/bin/sh
echo "Hello from a Flatpak Sandbox!"
```

3. Crie o ficheiro `manifest`, `pt.ua.deti.iei.HelloWorld.yml`. Este ficheiro define:
   - `app-id`: Um nome único, no formato DNS-reverso, para a sua aplicação.
   - `runtime` / `sdk`: O sistema base onde a sua aplicação irá correr e ser construída.
   - `command`: O programa a executar quando a aplicação arranca.
   - `modules`: A lista de passos de compilação. Aqui, definimos um módulo que instala o nosso `script` `hello.sh` no caminho executável da `sandbox` (`/app/bin/`).

```
app-id: pt.ua.deti.iei.HelloWorld
runtime: org.freedesktop.Platform
runtime-version: '25.08'
sdk: org.freedesktop.Sdk
command: hello.sh

modules:
  - name: hello-module
    buildsystem: simple
    build-commands:
      # Installs the script into the sandbox's /app/bin/ folder
      - install -Dm755 hello.sh /app/bin/hello.sh
    sources:
      # Tells the builder to find 'hello.sh' in our project dir
      - type: file
        path: hello.sh
```

4. **Construir o pacote:** Este comando executa o `flatpak-builder` com várias opções importantes.

- `--user`: Constrói e instala a aplicação apenas para o seu utilizador, sem necessitar de `sudo`.
- `--install`: Instala automaticamente a aplicação após uma compilação bem-sucedida.
- `--install-deps-from=flathub`: Encontra e instala automaticamente quaisquer SDKs ou `runtimes` em falta a partir do Flathub.
- `--force-clean`: Apaga o `build-dir` para garantir uma compilação limpa.
- `build-dir`: O nome do diretório temporário a usar para a compilação.

```
$ flatpak-builder --user --install --install-deps-from=flathub \
--force-clean build-dir pt.ua.deti.iei.HelloWorld.yml
```

5. **Executar e Limpar:** `flatpak run` executa a sua aplicação dentro da sua `sandbox`. Após usar, `cd ..` para sair do diretório.

```
$ flatpak run pt.ua.deti.iei.HelloWorld
$ flatpak uninstall --user pt.ua.deti.iei.HelloWorld
```

1.B: AppImage "Hello World"

O AppImage funciona ao empacotar um diretório inteiro (chamado `AppDir`).

1. Crie um diretório para este exercício:

```
$ mkdir ex1-appimage && cd ex1-appimage
```

2. Crie o `AppDir` e o `script` `AppRun` principal. O ficheiro `AppRun` é um `script` especial que atua como ponto de entrada (entrypoint). É a *primeira* coisa que é executada quando abre o AppImage. Também criamos um ficheiro `icon.png` vazio (dummy).

```
$ mkdir -p HelloWorld.AppDir
$ echo '#!/bin/sh' > HelloWorld.AppDir/AppRun
$ echo 'echo "Hello from an AppImage!"' >> HelloWorld.AppDir/AppRun
$ chmod +x HelloWorld.AppDir/AppRun
$ touch HelloWorld.AppDir/icon.png
```

3. Crie um ficheiro chamado `HelloWorld.AppDir/hello.desktop`. Este é um **ficheiro `.desktop`**, uma forma padrão de informar o ambiente de trabalho Linux sobre a sua aplicação. Ele define o `Name` (Nome) da aplicação, o comando a `Exec` (Executar) (o nosso `script` `AppRun`), e o `Icon` (Ícone) a usar. O `appimagetool` *exige* este ficheiro.

```
[Desktop Entry]
Name=Hello
Exec=AppRun
Icon=icon
Type=Application
Categories=Utility;
```

4. **Construir o pacote:** Executamos o `appimagetool` no nosso `AppDir`. Temos também de especificar `ARCH=x86_64` porque a ferramenta não consegue "adivinhar" a arquitetura a partir de um simples `script` de `shell`. Ela precisa disto para nomear o ficheiro final corretamente. Se necessário, altere a variável `ARCH` para `arm64`. Isto criará o ficheiro `Hello-x86_64.AppImage` ou `Hello-arm64.AppImage` em caso de sucesso.

```
$ ARCH=x86_64 appimagetool HelloWorld.AppDir
```

5. **Executar e Limpar:** Após usar, `cd ..` para sair do diretório.

```
$ chmod +x Hello-x86_64.AppImage
$ ./Hello-x86_64.AppImage

# Cleanup
$ rm -rf Hello-x86_64.AppImage
```

---

2. Aplicação CLI Python: Árvore ASCII 🌳

Vamos empacotar uma aplicação CLI (Command-Line Interface) simples em Python. Criaremos um `script` `pytree.py` que lista recursivamente diretórios num formato de árvore.

2.A: Executar com Virtual Environment (Venv)

Primeiro, vamos executar a aplicação nativamente para confirmar que funciona. Usaremos um **Python virtual environment** para gerir dependências, embora este `script` simples não tenha nenhuma.

Um `virtual environment` (venv) é uma "bolha" isolada para um projeto Python. Ele mantém o seu *próprio* interpretador Python e pacotes instalados, para que os pacotes deste projeto (ex: `pygame`) não entrem em conflito com os pacotes de outro projeto.

1. Crie um diretório de projeto:

```
$ mkdir ex2-pytree && cd ex2-pytree
```

2. Crie o `script` `pytree.py`, e torne-o executável: `chmod +x pytree.py`.

```
#!/usr/bin/env python3
import os
import sys

def tree(startpath):
    """Prints a directory tree."""
    for root, dirs, files in os.walk(startpath):
        # Don't visit .venv or __pycache__
        if '.venv' in dirs:
            dirs.remove('.venv')
        if '__pycache__' in dirs:
            dirs.remove('__pycache__')

        level = root.replace(startpath, '').count(os.sep)
        indent = '│   ' * (level - 1) + '├── ' if level > 0 else ''

        print(f'{indent}📁 {os.path.basename(root)}/')

        subindent = '│   ' * level + '├── '
        for f in files:
            print(f'{subindent}📄 {f}')

if __name__ == "__main__":
    # Use current directory or a specified path
    path = sys.argv[1] if len(sys.argv) > 1 else '.'
    tree(os.path.abspath(path))
```

3. **Executar a aplicação:** Como esta aplicação não tem dependências, podemos executá-la diretamente. Após usar, `cd ..` para sair do diretório.

```
$ ./pytree.py
# Tentar noutro diretório
$ ./pytree.py /tmp
```

2.B: Empacotar o `pytree` como um Flatpak

1. Crie um diretório de projeto:

```
$ mkdir ex2-flatpak && cd ex2-flatpak
```

2. Copie o ficheiro `pytree.py` do exercício anterior:

```
$ cp ../ex2-pytree/pytree.py .
```

3. Crie o `manifest` `pt.ua.deti.iei.pytree.yml`. Usamos o `org.gnome.Platform` como o nosso `runtime` porque ele inclui convenientemente um interpretador Python 3, pelo que não temos de construir o Python nós mesmos.

```
app-id: pt.ua.deti.iei.pytree
runtime: org.gnome.Platform
runtime-version: '48'
sdk: org.gnome.Sdk
command: pytree.py

modules:
  - name: pytree
    buildsystem: simple
    build-commands:
    - install -Dm755 pytree.py /app/bin/pytree.py
    sources:
    - type: file
      path: pytree.py
```

4. **Construir e Instalar:**

```
$ flatpak-builder --user --install --install-deps-from=flathub \
--force-clean build-dir pt.ua.deti.iei.pytree.yml
```

5. **Executar e Limpar:** Quando o executa pela primeira vez, ele lista apenas os ficheiros *dentro da sua própria sandbox*. Para o tornar útil, temos de lhe dar permissão para ver os nossos ficheiros do sistema anfitrião (host). `--filesystem=home` é um "portal" que abre um buraco na `sandbox`, dando à aplicação acesso ao nosso diretório home. Após usar, `cd ..` para sair do diretório.

```
$ flatpak run pt.ua.deti.iei.pytree

# Ele corre dentro de uma sandbox, por isso só se vê a si mesmo!
# Vamos dar-lhe acesso ao nosso diretório home para o testar:
$ flatpak run --filesystem=home pt.ua.deti.iei.pytree ~/

$ flatpak uninstall pt.ua.deti.iei.pytree
```

2.C: Empacotar o `pytree` como um AppImage

1. Crie um diretório de projeto:

```
$ mkdir ex2-appimage && cd ex2-appimage
```

2. Crie o `AppDir`:

```
$ mkdir -p Pytree.AppDir && cd Pytree.AppDir
```

3. **Descarregar e extrair o Python portátil:** Aqui, usamos o `wget` para descarregar uma versão do Python pré-construída e portátil. Um AppImage é apenas um sistema de ficheiros comprimido, por isso usamos `--appimage-extract` para o desempacotar. De seguida, movemos o seu conteúdo (`mv squashfs-root/* .`) para a raiz do nosso `AppDir`. Altere o URL do Python se usar outra arquitetura (como arm ou arm64).

```
$ wget "https://github.com/niess/python-appimage/releases/\
download/python3.10/python3.10.19-cp310-cp310-manylinux_2_28_x86_64.AppImage" \
-O python.AppImage
$ chmod +x python.AppImage
$ ./python.AppImage --appimage-extract
$ mv squashfs-root/* .
$ rm -rf python* squashfs-root/
```

3. **Copiar o seu `script`:** Copiamos o nosso `script` para o diretório `usr/bin` fornecido pelo Python portátil que acabámos de extrair.

```
$ cp ../../ex2-pytree/pytree.py usr/bin/
```

4. **Atualizar o ponto de entrada (entrypoint)** AppRun**:** O pacote Python portátil vem com o seu próprio `script` AppRun. Só precisamos de editar a sua *última linha* para chamar o nosso `script` `pytree.py` em vez de iniciar uma `shell` Python. Finalmente, torne-o executável: `chmod +x AppRun`

```
#! /bin/bash
# Se estiver a executar a partir de uma imagem extraída, então exporta ARGV0 e APPDIR
if [ -z "${APPIMAGE}" ]; then
    export ARGV0="$0"

    self=$(readlink -f -- "$0") # Proteger espaços (issue 55)
    here="${self%/*}"
    tmp="${here%/*}"
    export APPDIR="${tmp%/*}"
fi

# Resolver o comando de chamada (preservando links simbólicos).
export APPIMAGE_COMMAND=$(command -v -- "$ARGV0")

# Exportar TCl/Tk
export TCL_LIBRARY="${APPDIR}/usr/share/tcltk/tcl8.6"
export TK_LIBRARY="${APPDIR}/usr/share/tcltk/tk8.6"
export TKPATH="${TK_LIBRARY}"

# Exportar certificado SSL
export SSL_CERT_FILE="${APPDIR}/opt/_internal/certs.pem"

# Chamar o Python
"$APPDIR/opt/python3.10/bin/python3.10" "$APPDIR/usr/bin/pytree.py" "$@"
```

5. Crie um ficheiro chamado `pytree.desktop` e preencha-o. Também criamos um ficheiro `icon.png` vazio (dummy) para satisfazer o `appimagetool`.

```
[Desktop Entry]
Name=PyTree
Exec=AppRun
Icon=icon
Type=Application
Categories=Utility;
```

```
$ touch icon.png
```

6. **Construir, Executar e Limpar:** Após usar, `cd ..` para sair do diretório.

```
$ cd ..  # Voltar ao diretório ex2-appimage
$ ARCH=x86_64 appimagetool Pytree.AppDir

$ chmod +x PyTree-x86_64.AppImage
$ ./PyTree-x86_64.AppImage

# Testar no seu diretório home
$ ./PyTree-x86_64.AppImage ~/

$ rm -rf PyTree-x86_64.AppImage
```

---

3. Aplicação GUI Python: Jogo do Galo 🎮

3.A: Executar com Virtual Environment (venv)

Este passo simula o que um utilizador faria: descarregar o código-fonte, extraí-lo e executá-lo localmente.

1. Crie um diretório e descarregue o código-fonte: Este arquivo (um `.tar.gz`) contém uma pasta de nível superior. `tar --strip-components=1` é um comando útil para extrair o *conteúdo* dessa pasta diretamente para o nosso diretório atual, ignorando a própria pasta de nível superior.

```
$ mkdir ex3-tictactoe && cd ex3-tictactoe

$ wget "https://github.com/mariolpantunes/tictactoe/archive/refs/tags/tictactoe-1.0.tar.g
-O tictactoe-1.0.tar.gz

# Extrair o código-fonte descarregado
$ tar --strip-components=1 -zxvf tictactoe-1.0.tar.gz
```

2. **Criar e ativar o** `venv`**:** Desta vez, criar um `virtual environment` é crucial porque temos dependências. Deverá ver (`venv`) no início do 'prompt' do seu terminal. Isto significa que a sua `shell` está agora a usar o Python e o `pip` de dentro do diretório `./venv`.

```
$ python3 -m venv ./venv
$ source venv/bin/activate
```

3. **Instalar dependências a partir do ficheiro:** Um ficheiro `requirements.txt` lista todos os pacotes Python que um projeto necessita. `pip install -r` lê este ficheiro e instala-os (como o `pygame`) no `virtual environment` *ativo*.

```
$ pip install -r requirements.txt
```

4. **Executar o jogo:**

```
$ python main.py
```

5. **Desativar o venv:** Este comando restaura a sua `shell` para usar o Python padrão do sistema. Após usar, `cd ..` para sair do diretório.

```
$ deactivate
```

3.B: Empacotar o Jogo do Galo como um Flatpak

1. Crie um novo diretório para esta compilação:

```
$ mkdir ex3-flatpak && cd ex3-flatpak
```

2. **Crie o** `manifest` `pt.ua.deti.iei.tictactoe.yml`**:** Este `manifest` é mais complexo.
   - `finish-args`: Define o `PYTHONPATH` para que o interpretador Python dentro da `sandbox` possa encontrar o nosso módulo `minMaxAgent.py`, que instalamos em `/app/lib/game`.
   - módulo `python-deps`: Especifica manualmente o URL e o `checksum` (`sha256`) para o código-fonte do `pygame`. O `flatpak-builder` descarrega isto e constrói-o de raiz.
   - módulo `game`: Descarrega o código-fonte do jogo a partir do seu URL (tal como o `wget` fez). Os `build-commands` instalam então todas as partes do jogo: os `scripts` Python, a pasta `assets`, e os ficheiros `.desktop` e `icon` para integração no menu de aplicações.

```
app-id: pt.ua.deti.iei.tictactoe
runtime: org.gnome.Platform
runtime-version: "48"
sdk: org.gnome.Sdk
command: game
finish-args:
  - --share=ipc
  - --socket=x11
  - --socket=wayland
  - --device=dri
  - --env=PYTHONPATH=/app/lib/game
modules:
  - name: python-deps
    buildsystem: simple
    build-options:
      env:
        MAKEFLAGS: -j$(nproc)
    build-commands:
      - pip3 install --isolated --no-index --find-links="file://${PWD}" --prefix=/app pyg
```

```
    sources:
      - type: file
        url: https://pypi.io/packages/source/p/pygame/pygame-2.6.1.tar.gz
        sha256: 56fb02ead529cee00d415c3e007f75e0780c655909aaa8e8bf616ee09c9feb1f
  - name: game
    buildsystem: simple
    build-commands:
      - install -d /app/lib/game/
      - install -Dm644 minMaxAgent.py /app/lib/game/minMaxAgent.py
      - install -d /app/share/game/
      - cp -r assets /app/share/game/
      - install -Dm755 main.py /app/bin/game
      - install -Dm644 pt.ua.deti.iei.tictactoe.desktop
        /app/share/applications/pt.ua.deti.iei.tictactoe.desktop
      - install -Dm644 assets/icon.png
        /app/share/icons/hicolor/128×128/apps/pt.ua.deti.iei.tictactoe.png
    sources:
      - type: archive
        url: https://github.com/mariolpantunes/tictactoe/archive/refs/tags/tictactoe-1.0.
        sha256: 4210c04451ae8520770b0a7ab61e8b72f0ca46fbf2d65504d7d98646fda79b5a
```

4. **Construir e Instalar:** Após a instalação, o seu jogo deverá aparecer no menu de aplicações do seu desktop!

```
$ flatpak-builder --user --install --install-deps-from=flathub \
--force-clean build-dir pt.ua.deti.iei.tictactoe.yml
```

5. **Executar e Limpar:** Após usar, `cd ..` para sair do diretório.

```
$ flatpak run pt.ua.deti.iei.tictactoe
$ flatpak uninstall pt.ua.deti.iei.tictactoe
```

3.C: Empacotar o Jogo do Galo como um AppImage

1. Crie um diretório de compilação:

```
$ mkdir ex3-appimage && cd ex3-appimage
```

2. **Descarregar o código-fonte do jogo:**

```
$ wget "https://github.com/mariolpantunes/tictactoe/archive/refs/tags/tictactoe-1.0.tar.g
-O tictactoe-1.0.tar.gz
```

3. Crie o `AppDir`:

```
$ mkdir -p TTT.AppDir && cd TTT.AppDir
```

4. **Descarregar e extrair o Python portátil:** Isto é igual ao Exercício 2.C.

```
$ wget "https://github.com/niess/python-appimage/releases/\
download/python3.10/python3.10.19-cp310-cp310-manylinux_2_28_x86_64.AppImage" \
-O python.AppImage
$ chmod +x python.AppImage
$ ./python.AppImage --appimage-extract
$ mv squashfs-root/* .
$ rm -rf python* squashfs-root/
```

5. **Extrair o código-fonte do seu jogo:**

```
$ tar --strip-components=1 -zxvf ../tictactoe-1.0.tar.gz
```

6. **Instalar dependências a partir do `requirements.txt`:** Usamos o `pip` do Python *embutido* (`bundled`) para instalar pacotes. A `flag --target` diz ao `pip` para instalar o `pygame` *dentro* da pasta `site-packages` do nosso `AppDir`, e não no sistema anfitrião.

```
$ ./usr/bin/python3.10 -m pip install -r ./requirements.txt \
--target ./usr/lib/python3.10/site-packages/
```

7. **Copiar os ficheiros do seu jogo:** Movemos os `scripts` e `assets` do jogo para dentro do `AppDir`.

```
$ mv main.py minMaxAgent.py assets usr/bin/
```

8. **Atualizar o ponto de entrada (entrypoint)** `AppRun`**:** Este `script` `AppRun` é atualizado para definir a variável `PYTHONPATH`. Isto diz ao interpretador Python para procurar módulos em *dois* locais: o nosso diretório `site-packages` (para encontrar o `pygame`) e o nosso diretório `usr/bin` (para encontrar o `minMaxAgent.py`). Finalmente, torne-o executável: `chmod +x AppRun`.

```
#! /bin/bash
# Se estiver a executar a partir de uma imagem extraída, então exporta ARGV0 e APPDIR
if [ -z "${APPIMAGE}" ]; then
    export ARGV0="$0"

    self=$(readlink -f -- "$0") # Proteger espaços (issue 55)
    here="${self%/*}"
    tmp="${here%/*}"
    export APPDIR="${tmp%/*}"
fi

# Resolver o comando de chamada (preservando links simbólicos).
export APPIMAGE_COMMAND=$(command -v -- "$ARGV0")

# Exportar TCl/Tk
export TCL_LIBRARY="${APPDIR}/usr/share/tcltk/tcl8.6"
export TK_LIBRARY="${APPDIR}/usr/share/tcltk/tk8.6"
export TKPATH="${TK_LIBRARY}"

# Exportar certificado SSL
export SSL_CERT_FILE="${APPDIR}/opt/_internal/certs.pem"

# Exportar PyGame
export PYTHONPATH="$APPDIR/usr/lib/python3.10/site-packages:$APPDIR/usr/bin"

# Chamar o Python
"$APPDIR/opt/python3.10/bin/python3.10" "$APPDIR/usr/bin/main.py" "$@"
```

9. **Adicionar metadados:** Movemos o ficheiro `.desktop` e copiamos o ícone para a raiz do `AppDir` para que o `appimagetool` os possa encontrar.

```
$ mv pt.ua.deti.iei.tictactoe.desktop ./tictactoe.desktop
$ cp usr/bin/assets/icon.png ./pt.ua.deti.iei.tictactoe.png
```

10. **Construir, Executar e Limpar:**

```
$ cd .. # Voltar ao diretório ex3-appimage
$ appimagetool TTT.AppDir

$ chmod +x TicTacToe-x86_64.AppImage
$ ./TicTacToe-x86_64.AppImage

$ rm -rf *.AppImage tictactoe-v1.0.tar.gz
```

Conclusão

Nestes exercícios, empacotou uma aplicação Python de duas formas distintas: como um **AppImage** auto-suficiente e como um **Flatpak** em `sandbox`.

Embora ambos os métodos alcancem a portabilidade, este workshop destaca as vantagens significativas do ecossistema Flatpak, especialmente para aplicações complexas.

O processo **AppImage** exigiu que criássemos *manualmente* um `bundle`. Tivemos de:

1. Descarregar um interpretador Python portátil.

2. Instalar manualmente dependências numa pasta `site-packages` específica.
3. Escrever um `script` AppRun personalizado para definir variáveis de ambiente como o PYTHONPATH.

O processo **Flatpak**, em contraste, é **declarativo** (declarative) e **reprodutível** (reproducible).

1. **O 'Manifest' é a Receita:** Nós simplesmente *declaramos* todas as nossas necessidades num único ficheiro `manifest .yml`. Este ficheiro define a aplicação, as suas fontes (como o URL do GitHub), as suas dependências Python e as suas permissões de `sandbox`.
2. **Os 'Runtimes' são Eficientes:** Em vez de empacotar um interpretador Python de 100MB+, nós simplesmente requisitámos o `org.gnome.Platform`. Este `runtime` é descarregado *uma vez* pelo utilizador e partilhado por todas as suas aplicações Flatpak, tornando o pacote do nosso jogo incrivelmente pequeno e rápido de construir.
3. **A Compilação é Mais Fácil:** Não precisámos de escrever nenhuns `scripts` de `shell` complexos. O `flatpak-builder` tratou de todo o trabalho de descarregar o SDK, construir o `pygame`, e colocar os ficheiros nos diretórios corretos com base nos nossos simples comandos `install`.

No geral, o uso de `manifests` e `runtimes` partilhados pelo Flatpak resulta num processo de compilação que é muito mais automatizado, fácil de manter e eficiente, tanto para os developers como para os utilizadores finais.