Build a multi OS Python app in the cloud: PyInstaller on GitHub Actions

Motivation

Distributing Python apps for multiple OS types can be tricky. Especially, when you target a non-tech savvy audience. Delivering a single, stand-alone app often is the only viable approach. You can do this using PyInstaller. It packages everything your Python script needs to run, i.e. the interpreter and all dependencies, in a single file. For Windows this can be an .exe file and for MacOS an .app bundle. There is a limitation, though. To build your app for Windows, you need to run PyInstaller on Windows. This is the same for all OSes. Now, you have three options:

1. Stick to Linux, because who uses other OSes anyways?
2. Use a VirtualMachine to run the destination OS and build in there
3. Build in the cloud

Because the first option would only make for a very short post, let's go with the third one.

GitHub Actions

There are several options for building in the cloud. However, if you happen to use GitHub, GitHub Actions will be your best bet. It's a tool for CI/CD (continuous integration / deployment). It has seamless integration with your GitHub repository, is simple and free. You can make it execute actions on specific GitHub events. For that, you can define a workflow to be run on a customizable cloud machine that runs either Linux, Windows or MacOS. That makes it perfectly suitable for our use case, where we need to build on different platforms.
In what follows, I will omit explaining some of the more basic functionalities of GitHub Actions. Hence, you might need to take a look at the documentation when you get lost.

Creating a build workflow

We'll look at a hands-on, real-life example workflow to solve our challenge.
My open source app Clipster is a multi platform cloud clipboard solution. The desktop client is written in Go (there also is an older Python version) and is available on Linux, Windows and MacOS. To make it easy for users, the Windows and MacOS version is a single executable file. Hence, it is necessary to build those using PyInstaller on each corresponding OS for every new release.
The solution? On pushing a tagged commit to GitHub, each platform specific package is automatically built via GitHub Actions. Also, the resulting files are automatically added as assets to the new release on GitHub. This reduces the effort of releasing a new version to almost zero.
To get started with GitHub Actions, we create the following folder in our app's repo:

```
mkdir -p .github/workflows/
```

There, we add a `build.yml` file which defines our action. When pushing this file to the remote GitHub repo, the corresponding action will be automatically created. (You'll find a new workflow called "Build" in the `Actions` tab of the repo)
The file's content should look like this:

```
name: Build

on:
  push:
```

```yaml
    tags:
      - 'v*' # Push events to matching v*, i.e. v1.0, v20.15.10

jobs:

  createrelease:
    name: Create Release
    runs-on: [ubuntu-latest]
    steps:
    - name: Create Release
      id: create_release
      uses: actions/create-release@v1
      env:
        GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
      with:
        tag_name: ${{ github.ref }}
        release_name: Release ${{ github.ref }}
        draft: false
        prerelease: false
    - name: Output Release URL File
      run: echo "${{ steps.create_release.outputs.upload_url }}" > release_url.txt
    - name: Save Release URL File for publish
      uses: actions/upload-artifact@v1
      with:
        name: release_url
        path: release_url.txt

  build:
    name: Build packages
    needs: createrelease
    runs-on: ${{ matrix.os }}
    strategy:
      matrix:
        include:
          - os: macos-latest
            TARGET: macos
            CMD_BUILD: >
                pyinstaller -F -w -n clipster -i resources/clipster.icns cli.py &&
                cd dist/ &&
                zip -r9 clipster clipster.app/
            OUT_FILE_NAME: clipster.zip
            ASSET_MIME: application/zip
          - os: windows-latest
            TARGET: windows
            CMD_BUILD: pyinstaller -F -w -n clipster -i resources/clipster.ico cli.py
            OUT_FILE_NAME: clipster.exe
            ASSET_MIME: application/vnd.microsoft.portable-executable
    steps:
    - uses: actions/checkout@v1
    - name: Set up Python 3.8
      uses: actions/setup-python@v2
      with:
        python-version: 3.8
    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        pip install -r requirements.txt
    - name: Build with pyinstaller for ${{matrix.TARGET}}
      run: ${{matrix.CMD_BUILD}}
    - name: Load Release URL File from release job
      uses: actions/download-artifact@v1
      with:
        name: release_url
    - name: Get Release File Name & Upload URL
      id: get_release_info
      shell: bash
      run: |
        value=`cat release_url/release_url.txt`
        echo ::set-output name=upload_url::$value
    - name: Upload Release Asset
      id: upload-release-asset
      uses: actions/upload-release-asset@v1
      env:
        GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
      with:
        upload_url: ${{ steps.get_release_info.outputs.upload_url }}
        asset_path: ./dist/${{ matrix.OUT_FILE_NAME}}
        asset_name: ${{ matrix.OUT_FILE_NAME}}
        asset_content_type: ${{ matrix.ASSET_MIME}}
```

First, we define that the following jobs are to be run on each push event. But only, if the commit is tagged using "v" as a prefix. The `createrelease` job takes care of creating a new release on GitHub. For that, it uses the `create-release` action in its first step. An action is a collection of commands and routines that achieves a specific target. For common objectives there are pre-defined and user-defined actions that you can use.
The second and third steps are a bit more tricky:

```
    [...]
    - name: Output Release URL File
      run: echo "${{ steps.create_release.outputs.upload_url }}" > release_url.txt
    - name: Save Release URL File for publish
      uses: actions/upload-artifact@v1
      with:
        name: release_url
        path: release_url.txt
    [...]
```

For the upcoming jobs, we'll need to upload files as assets to the newly created release on GitHub. For that, there is a specific URL location which must be used. That's what the first step does. It gets this location by reading the content of the environment variable `steps.create_release.outputs.upload_url` and writes it to the file `release_url.txt`. Lastly, we upload this text file (again using a pre-defined action) so that we can extract the URL from it later on.

Next, we look at the `build` job. This is only run after `createrelease` has finished. Here, it gets a little bit more tricky. Inside our `strategy` part, we define two different "versions" of a step. The first one will be run on MacOS. It will execute `pyinstaller` to build the MacOS package of our script. The second one will be run on a Windows machine and again use `pyinstaller` to create the Windows `.exe` version of the app.
After defining this strategy, the following steps are run:

1. Using `actions/checkout@v1`, the repo is cloned to the cloud machine
2. The `requirements.txt` in the repo is used to resolve the dependencies of our app
3. The packages are created using PyInstaller, according to the strategy defined beforehand (this step is run once for each TARGET)

The next parts take a little bit more time to figure out:

```
    - name: Load Release URL File from release job
      uses: actions/download-artifact@v1
      with:
        name: release_url
    - name: Get Release File Name & Upload URL
      id: get_release_info
      shell: bash
      run: |
        value=`cat release_url/release_url.txt`
        echo ::set-output name=upload_url::$value
    - name: Upload Release Asset
      id: upload-release-asset
      uses: actions/upload-release-asset@v1
      env:
        GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
      with:
        upload_url: ${{ steps.get_release_info.outputs.upload_url }}
        asset_path: ./dist/${{ matrix.OUT_FILE_NAME}}
        asset_name: ${{ matrix.OUT_FILE_NAME}}
        asset_content_type: ${{ matrix.ASSET_MIME}}
```

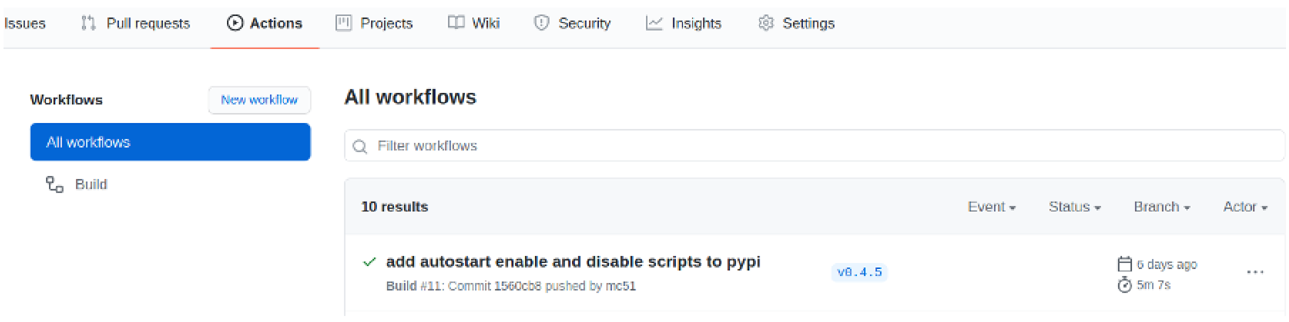
First, we download the file containing our release URL that we created in the `createrelease` job just before.
Then, using bash, we extract the URL from the file and save it to a variable.
Finally, we upload the packages created in the steps before, i.e. the MacOS and Windows package of our app. For that, we use the URL stored in the variable from before, which points at the location where we can upload assets for the release created before.

Results of the build workflow

As mentioned before, the workflow is triggered when you push a new commit to the remote repo.
Here's an example:

```
git commit -a -m "add autostart enable and disable scripts to pypi"
git tag v0.4.5
git push
git push origin --tags
```

After the push, the build workflow will be automatically triggered. You can find its status under the `Actions` tab in your repo:

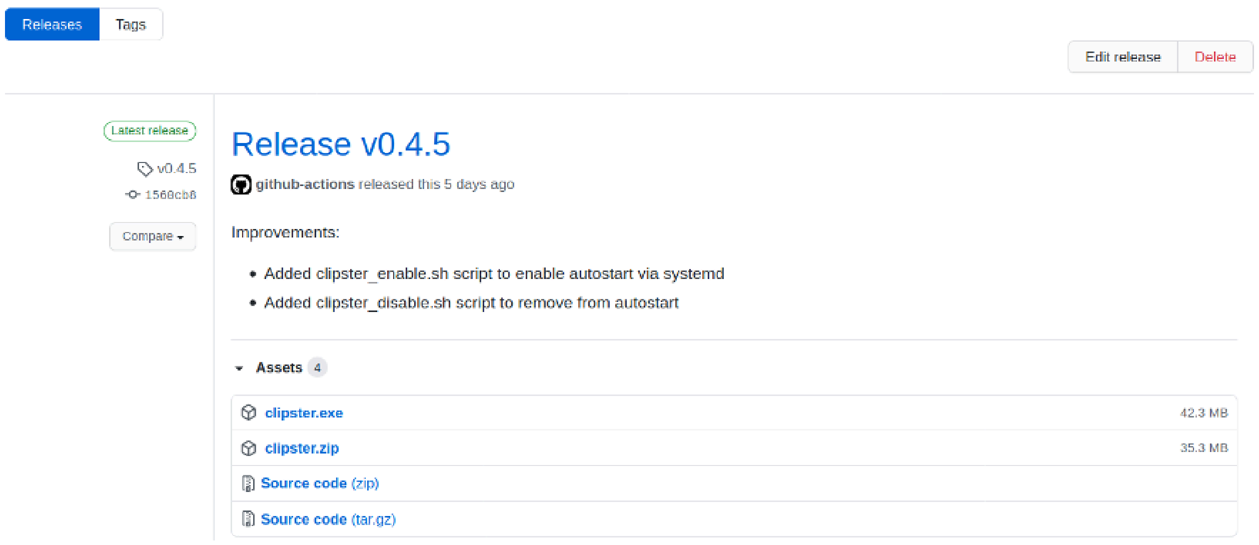


10 results

add autostart enable and disable scripts to pypi — v0.4.5 — 6 days ago — 5m 7s
Build #11: Commit 1560cb8 pushed by mc51

Figure 1. Status after a successful GitHub Actions build

There, you can also go into more depth and check the details of your build, e.g. see what commands are executed on the cloud machine. This is helpful when there are issues with the workflow.
In our case, the build was successful. Hence, the new release has been created and the packages for MacOS and Windows have been added as assets to it:



Release v0.4.5

github-actions released this 5 days ago

Improvements:

- Added clipster_enable.sh script to enable autostart via systemd
- Added clipster_disable.sh script to remove from autostart

Assets 4

clipster.exe	42.3 MB
clipster.zip	35.3 MB
Source code (zip)	
Source code (tar.gz)	

Figure 2. Release with packages as assets

You can always refers to the latest release of an asset file by using the following location:
https://github.com/username/Repo-Name/releases/latest/download/file.zip

Summary

Building in the cloud is super convenient and rather simple. Moreover, you can build for different OS types. This makes simultaneously releasing apps for multiple platforms a breeze. We’ve seen how to setup GitHub Actions to perform a cloud build of a Python app using PyInstaller. It takes care of creating a new release, building the OS specific packages and uploading them to the release page as assets.