DevOps with GitHub Actions – Lesson 3

Facilitated by Kent State University Topic: Packaging Applications for Deployment (Artifacts, Wheels, Bundles) Duration: 1 Hour

Learning Objectives

By the end of this lesson, participants will be able to:

- Package Python applications using wheels or bundled directories
- Create build artifacts in GitHub Actions and store them for later jobs
- Use caching to speed up builds
- Understand how packaging fits into deployment pipelines
- Version packages properly for release management
- Troubleshoot common packaging issues

I. Why Packaging Matters (5 minutes)

Before deploying code, it should be packaged in a stable, reproducible format. Packaging simplifies:

- Dependency installation
- Reproducible builds
- Versioned releases
- Artifact storage and reuse

Common package types:

- **Python wheels (.whl)**: Binary distribution format for Python packages
- **Tarball bundles (.tar.gz)**: Source distribution, platform-independent
- **Static folders**: For simple apps or frontend builds
- **Container images**: For complex applications (covered in later lessons)

When to Use Each Format:

Use Python Wheels when:

- Distributing Python libraries or applications
- You want faster installation than source distributions
- Your code includes compiled extensions
- You're publishing to PyPI or a private package index

Use Bundles/Tarballs when:

- Packaging web applications (HTML/CSS/JS)
- Creating releases of non-Python applications
- You need platform-independent source code
- Bundling configuration files with your application

Use Static Folders when:

- Deploying simple static sites
- Your app has no build step
- Files can be served directly (e.g., documentation)

II. Creating Python Wheels (15 minutes)

A. Project structure example

```
myapp/
    myapp/
        __init__.py
        app.py
    pyproject.toml
    README.md
```

B. Complete pyproject.toml with versioning

```toml
[build-system]
requires = ["setuptools>=61.0"]
build-backend = "setuptools.build_meta"

[project]
name = "myapp"
version = "0.1.0"  # Semantic versioning: MAJOR.MINOR.PATCH
description = "My application package"
readme = "README.md"
requires-python = ">=3.8"
dependencies = [
    "requests>=2.28.0",
    "click>=8.0.0",
]

[project.optional-dependencies]
dev = [
    "pytest>=7.0.0",
    "black>=22.0.0",
]
```

Versioning Best Practices:

- Use semantic versioning (SemVer): MAJOR.MINOR.PATCH
- MAJOR: Breaking changes
- MINOR: New features (backward compatible)
- PATCH: Bug fixes
- Use version tags in git: `v0.1.0`, `v1.0.0`, etc.

C. Build command

```
python -m build
```

This creates:

- `dist/myapp-0.1.0-py3-none-any.whl` (wheel - binary distribution)
- `dist/myapp-0.1.0.tar.gz` (source distribution)

D. Common Packaging Issues and Solutions

Problem: "No module named 'build'"

```
# Solution: Install build tools first
pip install build
```

Problem: "Package doesn't include my data files"

```
# Solution: Add to pyproject.toml
[tool.setuptools.package-data]
myapp = ["data/*.json", "templates/*.html"]
```

Problem: "Version conflicts with dependencies"

```
# Solution: Pin compatible versions
dependencies = [
    "requests>=2.28.0,<3.0.0",  # Allow minor updates, prevent breaking changes
]
```

Problem: "Build fails with 'no such file or directory'"

```
# Solution: Check your project structure
tree myapp/  # Verify files exist
cat pyproject.toml  # Check paths are correct
```

III. Packaging in GitHub Actions with Caching (15 minutes)

A. Basic Build Workflow

```
name: Build Package
on: [push]

jobs:
```

```yaml
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Set up Python
        uses: actions/setup-python@v4
        with:
          python-version: "3.11"

      - name: Cache pip dependencies
        uses: actions/cache@v3
        with:
          path: ~/.cache/pip
          key: ${{ runner.os }}-pip-${{ hashFiles('**/pyproject.toml') }}
          restore-keys: |
            ${{ runner.os }}-pip-

      - name: Install build tools
        run: pip install build

      - name: Build wheel
        run: python -m build

      - name: Upload artifact
        uses: actions/upload-artifact@v3
        with:
          name: app-wheel
          path: dist/*.whl
```

B. Advanced: Version from Git Tags

```yaml
name: Build and Release
on:
  push:
    tags:
      - 'v*'  # Trigger on version tags like v1.0.0

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Get version from tag
        id: get_version
        run: echo "VERSION=${GITHUB_REF#refs/tags/v}" >> $GITHUB_OUTPUT

      - name: Set up Python
        uses: actions/setup-python@v4
        with:
          python-version: "3.11"

      - name: Update version in pyproject.toml
        run: |
          sed -i "s/version = .*/version = \"${{
steps.get_version.outputs.VERSION }}\"/" pyproject.toml

      - name: Build wheel
        run: |
          pip install build
          python -m build

      - name: Upload release artifacts
        uses: actions/upload-artifact@v3
        with:
          name: release-${{ steps.get_version.outputs.VERSION }}
          path: dist/*
```

C. Understanding Caching

Why cache?

- Speeds up builds by reusing downloaded dependencies
- Reduces network usage
- Makes builds more reliable

What to cache:

- pip packages: `~/.cache/pip`
- npm packages: `node_modules`
- Build artifacts from previous steps

Cache key strategy:

- Use hash of dependency files (e.g., `pyproject.toml`)
- Include OS and language version
- Cache invalidates when dependencies change

IV. Bundling Non-Python Apps (10 minutes)

A. Creating ZIP bundles

```
- name: Create bundle
  run: zip -r app.zip ./src

- name: Upload bundle
  uses: actions/upload-artifact@v3
  with:
    name: app-bundle
    path: app.zip
```

B. Creating TAR.GZ bundles (better for Unix/Linux)

```
- name: Create tarball
  run: tar -czf app.tar.gz ./src

- name: Upload tarball
  uses: actions/upload-artifact@v3
  with:
    name: app-tarball
    path: app.tar.gz
```

C. Bundling with version information

```
- name: Get short commit hash
  id: hash
  run: echo "SHA_SHORT=$(git rev-parse --short HEAD)" >> $GITHUB_OUTPUT

- name: Create versioned bundle
  run: |
    VERSION=$(date +%Y%m%d)-${{ steps.hash.outputs.SHA_SHORT }}
    tar -czf myapp-${VERSION}.tar.gz ./src
    echo "Bundle created: myapp-${VERSION}.tar.gz"

- name: Upload bundle
  uses: actions/upload-artifact@v3
  with:
    name: app-bundle-${{ steps.hash.outputs.SHA_SHORT }}
    path: "*.tar.gz"
```

D. Common Bundling Issues

Problem: Large bundle size

```
# Solution: Exclude unnecessary files
zip -r app.zip ./src -x "*.pyc" "*.git*" "__pycache__/*"
# Or use .gitignore-like exclusions
```

Problem: Permission issues after extraction

```
# Solution: Preserve permissions with tar
tar -czpf app.tar.gz ./src  # Note the 'p' flag
```

V. Using Artifacts Across Jobs (10 minutes)

A. Complete multi-job workflow

```yaml
name: Build and Deploy
on: [push]

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Set up Python
        uses: actions/setup-python@v4
        with:
          python-version: "3.11"

      - name: Cache dependencies
        uses: actions/cache@v3
        with:
          path: ~/.cache/pip
          key: ${{ runner.os }}-pip-${{ hashFiles('**/pyproject.toml') }}

      - name: Build wheel
        run: |
          pip install build
          python -m build

      - name: Upload wheel artifact
        uses: actions/upload-artifact@v3
        with:
          name: wheel
          path: dist/*.whl
          retention-days: 7  # How long to keep the artifact

  test:
    runs-on: ubuntu-latest
    needs: build  # Wait for build job to complete
    steps:
      - uses: actions/checkout@v3

      - name: Download wheel artifact
        uses: actions/download-artifact@v3
        with:
          name: wheel

      - name: Install and test wheel
        run: |
          pip install *.whl
          python -m pytest tests/

  deploy:
    runs-on: ubuntu-latest
    needs: [build, test]  # Wait for both build and test
    if: github.ref == 'refs/heads/main'  # Only deploy from main branch
    steps:
      - uses: actions/download-artifact@v3
        with:
          name: wheel

      - name: Deploy wheel
        run: |
          echo "Deploying wheel to production..."
          # Example: Upload to PyPI, deploy to server, etc.
```

B. Artifact retention and storage

```
- name: Upload artifact with custom retention
  uses: actions/upload-artifact@v3
  with:
    name: my-artifact
    path: dist/
    retention-days: 30  # Keep for 30 days (max 90)
```

Artifact Best Practices:

- Set appropriate retention periods (default is 90 days)
- Use descriptive artifact names
- Keep artifacts small (GitHub has storage limits)
- Clean up old artifacts periodically

C. Troubleshooting Artifacts

Problem: "Artifact not found" in downstream job

```
# Solution: Ensure 'needs' dependency is set correctly
jobs:
  download-job:
    needs: upload-job  # Must specify dependency
    steps:
      - uses: actions/download-artifact@v3
```

Problem: Multiple files with same name

```
# Solution: Use unique artifact names
- uses: actions/upload-artifact@v3
  with:
    name: wheel-${{ matrix.python-version }}  # Include matrix variable
    path: dist/*.whl
```

VI. Exercise (5 minutes)

Task: Package and Test a Python Project

1. Create a simple Python project with this structure:

   ```
   calculator/
       calculator/
           __init__.py
           operations.py
       tests/
           test_operations.py
       pyproject.toml
   ```

2. Add a `pyproject.toml` with:
   - Project name and version
   - Dependencies (if any)
   - Build system configuration

3. Create a GitHub Actions workflow that:
   - Builds a wheel from your project
   - Uploads the wheel as an artifact
   - Downloads the artifact in a separate job
   - Installs and tests the wheel

4. Bonus: Add caching for pip dependencies

Solution Hints:

pyproject.toml:

```toml
[build-system]
requires = ["setuptools>=61.0"]
build-backend = "setuptools.build_meta"

[project]
name = "calculator"
version = "1.0.0"
requires-python = ">=3.8"
```

Workflow structure:

```yaml
name: Build and Test
on: [push]
jobs:
```

```
  build:
    # Add build steps here
  test:
    needs: build
    # Add test steps here
```

VII. Hands-On Learning: SimpleStat Package (10 minutes)

A. Overview of the Example Project

In the `topic-03-packaging/wheels/` directory, you'll find a complete example package called **SimpleStat** that demonstrates all the concepts from this lesson. This is a working Python package that you can build, test, and install.

Project Structure:

```
wheels/
    simplestat/
        __init__.py       # Package initialization and exports
        stats.py          # Statistical functions implementation
    pyproject.toml        # Package configuration and metadata
    README.md             # Documentation and usage examples
    test_simplestat.py    # Test script to verify functionality
```

B. What SimpleStat Demonstrates

The SimpleStat package includes:

1. **Six statistical functions:**
    - `mean()` - Calculate average of numbers
    - `median()` - Find middle value
    - `mode()` - Find most common value
    - `variance()` - Measure data spread
    - `standard_deviation()` - Square root of variance
    - `range_of_values()` - Difference between max and min
2. **Professional packaging elements:**
    - Type hints for better IDE support
    - Comprehensive docstrings with examples
    - Error handling for edge cases
    - Proper semantic versioning (1.0.0)
    - Complete pyproject.toml configuration
3. **Testing and validation:**
    - Test script that verifies all functions
    - Error handling tests

  - Example usage patterns

C. Learning Exercises with SimpleStat

Exercise 1: Build the Package

```
cd topic-03-packaging/wheels/
python -m build
```

This creates wheel and source distributions in the `dist/` directory.

Exercise 2: Test Before Installing

```
python test_simplestat.py
```

This runs all tests to ensure the package works correctly.

Exercise 3: Install and Use

```
pip install dist/simplestat-1.0.0-py3-none-any.whl
python -c "from simplestat import mean; print(mean([1, 2, 3, 4, 5]))"
```

Exercise 4: Modify and Rebuild Try making changes to demonstrate versioning:

1. Add a new function to `stats.py` (e.g., `geometric_mean()`)
2. Update the version in `pyproject.toml` to `1.1.0` (minor version bump for new feature)
3. Rebuild: `python -m build`
4. Notice the new wheel has the updated version number

Exercise 5: Create a GitHub Actions Workflow Create `.github/workflows/build-simplestat.yml` to:

- Build the SimpleStat package
- Run the test script
- Upload the wheel as an artifact
- Add caching for pip dependencies

Example workflow starter:

```
name: Build SimpleStat
on: [push]

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Set up Python
        uses: actions/setup-python@v4
        with:
          python-version: "3.11"

      - name: Cache pip
        uses: actions/cache@v3
        with:
          path: ~/.cache/pip
          key: ${{ runner.os }}-pip-${{ hashFiles('**/pyproject.toml') }}

      - name: Install build tools
        run: pip install build

      - name: Run tests
        run: python topic-03-packaging/wheels/test_simplestat.py

      - name: Build package
        run: |
          cd topic-03-packaging/wheels/
          python -m build

      - name: Upload wheel
        uses: actions/upload-artifact@v3
        with:
          name: simplestat-wheel
          path: topic-03-packaging/wheels/dist/*.whl
```

D. Extension Challenges

Once you're comfortable with SimpleStat, try these challenges:

1. **Add Dependencies:** Modify SimpleStat to use numpy for calculations, update `pyproject.toml` accordingly
2. **Create Multiple Distributions:** Build for different Python versions using a matrix strategy in GitHub Actions
3. **Add Entry Points:** Create a command-line interface that can be run as `simplestat-cli`
4. **Publish to Test PyPI:** Practice publishing to Test PyPI (test.pypi.org)
5. **Version Automation:** Create a workflow that automatically bumps versions based on commit messages or tags

E. Key Learning Points from SimpleStat

By working with this example, you'll understand:

- **Package Structure:** How to organize Python code as an installable package

- **Metadata Configuration:** What goes in pyproject.toml and why
- **Build Process:** How `python -m build` creates distributable packages
- **Testing:** How to validate packages before distribution
- **Versioning:** When to bump major, minor, or patch versions
- **Documentation:** How to write effective READMEs and docstrings
- **Workflow Integration:** How to automate building and testing in CI/CD

Why This Matters: SimpleStat is a minimal but complete example. Real-world packages follow the same patterns - they're just larger. Understanding this simple package prepares you to work with or create any Python package for deployment.

VIIa. Tarball Packaging: C Statistics Calculator (10 minutes)

A. Overview of the C Example Project

In the `topic-03-packaging/tarball/` directory, you'll find a **C statistics calculator** that demonstrates tarball packaging for compiled applications. This shows how to package non-Python projects for distribution.

Project Structure:

```
tarball/
    stats.h            # Header file with function declarations
    stats.c            # Statistics functions implementation
    main.c             # Demo program
    Makefile           # Build automation
    README.md          # Documentation
    .gitignore         # Exclude build artifacts
```

B. What the C Project Demonstrates

The C statistics calculator includes:

1. **Seven statistical functions:**
    - `mean()` - Calculate average
    - `median()` - Find middle value (handles sorting)
    - `variance()` - Measure spread (sample or population)
    - `standard_deviation()` - Square root of variance
    - `range_values()` - Difference between max and min
    - `min_value()` / `max_value()` - Find extremes
2. **Professional C development practices:**
    - Header/implementation file separation
    - Error handling for edge cases
    - Memory management (malloc/free in median)
    - Makefile with multiple targets
    - Documentation with usage examples

3. **Build automation:**
    - Compile with optimization flags
    - Clean build artifacts
    - Install/uninstall targets
    - Help documentation

C. Building and Testing the C Application

Build the application:

```
cd topic-03-packaging/tarball/
make
```

Run the demo:

```
./stats_calc
```

Clean build artifacts:

```
make clean
```

D. Creating Tarballs for Distribution

Basic tarball creation:

```
cd topic-03-packaging/
tar -czf stats-calc-1.0.0.tar.gz tarball/
```

This creates a gzipped tarball (`.tar.gz`) containing all source files.

Excluding build artifacts:

```
tar -czf stats-calc-1.0.0.tar.gz --exclude='*.o' --exclude='stats_calc'
tarball/
```

Better structure with version in path:

```
cd topic-03-packaging/
cp -r tarball stats-calc-1.0.0
```

```
tar -czf stats-calc-1.0.0.tar.gz stats-calc-1.0.0/
rm -rf stats-calc-1.0.0
```

This creates a tarball where files extract to `stats-calc-1.0.0/` directory.

E. GitHub Actions Workflow for C Projects

Example workflow for building and packaging C applications:

```
name: Build C Stats Calculator
on: [push]

jobs:
  build-and-package:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Install build tools
        run: sudo apt-get update && sudo apt-get install -y build-
essential

      - name: Build application
        run: |
          cd topic-03-packaging/tarball/
          make

      - name: Test application
        run: |
          cd topic-03-packaging/tarball/
          ./stats_calc

      - name: Create source tarball
        run: |
          cd topic-03-packaging/
          cp -r tarball stats-calc-1.0.0
          tar -czf stats-calc-1.0.0.tar.gz \
            --exclude='*.o' \
            --exclude='stats_calc' \
            stats-calc-1.0.0/
          rm -rf stats-calc-1.0.0

      - name: Upload tarball artifact
        uses: actions/upload-artifact@v3
        with:
          name: stats-calc-tarball
          path: topic-03-packaging/stats-calc-1.0.0.tar.gz

      - name: Upload binary
        uses: actions/upload-artifact@v3
        with:
          name: stats-calc-binary-linux
          path: topic-03-packaging/tarball/stats_calc
```

F. Multi-Platform Builds with Matrix Strategy

For cross-platform applications, build on multiple operating systems:

```yaml
name: Multi-Platform Build
on: [push]

jobs:
  build:
    strategy:
      matrix:
        os: [ubuntu-latest, macos-latest]
    runs-on: ${{ matrix.os }}

    steps:
      - uses: actions/checkout@v3

      - name: Build
        run: |
          cd topic-03-packaging/tarball/
          make

      - name: Upload binary
        uses: actions/upload-artifact@v3
        with:
          name: stats-calc-${{ matrix.os }}
          path: topic-03-packaging/tarball/stats_calc
```

G. Extracting and Using Tarballs

Users download and extract:

```bash
tar -xzf stats-calc-1.0.0.tar.gz
cd stats-calc-1.0.0/
```

Build from source:

```bash
make
./stats_calc
```

Install system-wide (optional):

```
sudo make install
```

H. Key Differences: Wheels vs Tarballs

Aspect	Python Wheels	Tarballs
Language	Python-specific	Language-agnostic
Installation	`pip install`	Manual extraction + build
Dependencies	Managed by pip	User must install separately
Compilation	Pre-compiled or none	User compiles from source
Use Case	Python packages	Any source code project
Distribution	PyPI or private index	Direct download, GitHub releases

I. Learning Exercises with C Stats Calculator

Exercise 1: Build from Source

```
cd topic-03-packaging/tarball/
make
./stats_calc
```

Exercise 2: Create Your Own Tarball

```
cd topic-03-packaging/
tar -czf mystats-1.0.0.tar.gz --exclude='*.o' --exclude='stats_calc'
tarball/
tar -tzf mystats-1.0.0.tar.gz  # List contents
```

Exercise 3: Simulate User Experience

```
# Extract in a temporary location
mkdir /tmp/test-install
cd /tmp/test-install
tar -xzf /path/to/stats-calc-1.0.0.tar.gz
cd stats-calc-1.0.0/
make
./stats_calc
```

Exercise 4: Add to GitHub Actions Create a workflow that:

- Builds the C application

- Runs it to verify it works
- Creates a tarball
- Uploads both source tarball and compiled binary as artifacts
- Uses matrix strategy to build on Ubuntu and macOS

Exercise 5: Create a Release On version tags (e.g., `v1.0.0`):

- Build on multiple platforms
- Create platform-specific tarballs
- Upload to GitHub Releases

J. Extension Challenges

1. **Add Unit Tests:** Create a test harness in C that validates each function
2. **Cross-Compilation:** Build for different architectures (x86_64, ARM)
3. **Static Linking:** Create a standalone binary with no external dependencies
4. **Package Metadata:** Add a VERSION file or embed version in the binary
5. **Autotools:** Convert to use `autoconf`/`automake` for better portability

K. Why Both Examples Matter

SimpleStat (Python Wheel):

- Modern Python packaging
- Managed dependencies
- Easy installation
- Platform-independent (usually)

Stats Calculator (C Tarball):

- Traditional source distribution
- Cross-platform compilation
- No package manager required
- Shows packaging for compiled languages

Real-World Application: Many projects use both approaches - Python packages might include C extensions that are compiled during installation. Understanding both packaging methods prepares you for diverse deployment scenarios.

VIIb. Enterprise Package Distribution (8 minutes)

A. Why Companies Need Private Package Repositories

GitHub Actions artifacts are temporary (max 90 days) and only available within workflows. For production deployment, companies need:

- **Long-term storage** of versioned releases
- **Access control** for proprietary code
- **Dependency management** across teams
- **Security scanning** and compliance
- **Promotion workflows** (dev → staging → prod)

- **Single source of truth** for all artifacts

B. Package Repository Solutions

1. For Python Packages (Wheels)

Companies host private PyPI-compatible repositories:

Enterprise Solutions:

- **AWS CodeArtifact** - Managed service, integrates with AWS
- **Azure Artifacts** - Microsoft cloud solution
- **Google Artifact Registry** - GCP solution
- **JFrog Artifactory** - Multi-format, on-prem or cloud
- **Sonatype Nexus** - Multi-format repository manager

Self-Hosted (Lightweight):

- **devpi** - Python-specific, simple to set up
- **PyPI Server** - Minimal PyPI implementation
- **Simple Index** - Just static files + web server

2. For Tarballs and Binaries

- Same enterprise tools (Artifactory, Nexus)
- Generic artifact repositories
- Cloud storage (S3, Azure Blob, GCS)
- GitHub/GitLab Releases for open projects

C. Private PyPI Repository Structure

Simple Index Layout (PEP 503):

```
https://pypi.company.com/
└── simple/
    ├── index.html              # Lists all packages
    ├── simplestat/
    │   ├── index.html          # Lists simplestat versions
    │   ├── simplestat-1.0.0-py3-none-any.whl
    │   ├── simplestat-1.1.0-py3-none-any.whl
    │   └── simplestat-2.0.0-py3-none-any.whl
    └── myapp/
        ├── index.html
        └── myapp-0.1.0-py3-none-any.whl
```

Installing from private repository:

```
# One-time install
pip install --index-url https://pypi.company.com/simple/ simplestat
```

```
# Configure pip permanently
pip config set global.index-url https://pypi.company.com/simple/

# In requirements.txt
--index-url https://pypi.company.com/simple/
simplestat==1.0.0
requests>=2.28.0
```

D. Enterprise Tarball Repository Structure

Organized by project and version:

```
https://releases.company.com/
├── stats-calc/
│   ├── 1.0.0/
│   │   ├── stats-calc-1.0.0.tar.gz
│   │   ├── stats-calc-1.0.0-linux-x86_64.tar.gz
│   │   ├── stats-calc-1.0.0-linux-arm64.tar.gz
│   │   ├── stats-calc-1.0.0-macos-x86_64.tar.gz
│   │   ├── stats-calc-1.0.0-macos-arm64.tar.gz
│   │   └── checksums.sha256
│   └── 2.0.0/
│       └── ...
└── other-project/
    └── ...
```

E. Publishing to Private Repositories from GitHub Actions

Publishing Python wheels to private PyPI:

```
name: Build and Publish
on:
  push:
    tags:
      - 'v*'

jobs:
  build-and-publish:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Set up Python
        uses: actions/setup-python@v4
        with:
          python-version: "3.11"

      - name: Build package
        run: |
          pip install build
          python -m build

      - name: Publish to company PyPI
        env:
          TWINE_USERNAME: ${{ secrets.PYPI_USERNAME }}
          TWINE_PASSWORD: ${{ secrets.PYPI_TOKEN }}
          TWINE_REPOSITORY_URL: https://pypi.company.com/upload/
        run: |
          pip install twine
          twine upload dist/*
```

Publishing tarballs to artifact server:

```
name: Build and Upload Tarball
on:
  push:
    tags:
      - 'v*'

jobs:
  build-and-upload:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Get version from tag
        id: version
        run: echo "VERSION=${GITHUB_REF#refs/tags/v}" >> $GITHUB_OUTPUT

      - name: Build application
        run: |
          cd topic-03-packaging/tarball/
          make

      - name: Create tarball
        run: |
          cd topic-03-packaging/
          cp -r tarball stats-calc-${{ steps.version.outputs.VERSION }}
          tar -czf stats-calc-${{ steps.version.outputs.VERSION }}.tar.gz
\
            --exclude='*.o' --exclude='stats_calc' \
            stats-calc-${{ steps.version.outputs.VERSION }}/

      - name: Upload to artifact server
        env:
          ARTIFACT_TOKEN: ${{ secrets.ARTIFACT_TOKEN }}
        run: |
          VERSION=${{ steps.version.outputs.VERSION }}
          curl -H "Authorization: Bearer ${ARTIFACT_TOKEN}" \
               --upload-file stats-calc-${VERSION}.tar.gz \
              https://releases.company.com/stats-calc/${VERSION}/stats-
calc-${VERSION}.tar.gz
```

F. AWS CodeArtifact Example

Setting up AWS CodeArtifact authentication:

```
name: Publish to AWS CodeArtifact
on:
  push:
    tags:
      - 'v*'

jobs:
  publish:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Configure AWS credentials
        uses: aws-actions/configure-aws-credentials@v2
        with:
          aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
          aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
          aws-region: us-east-1

      - name: Set up Python
        uses: actions/setup-python@v4
        with:
          python-version: "3.11"

      - name: Get CodeArtifact token
        run: |
          export CODEARTIFACT_TOKEN=$(aws codeartifact get-authorization-
token \
            --domain my-company \
            --domain-owner 123456789012 \
            --query authorizationToken \
            --output text)
          echo "::add-mask::$CODEARTIFACT_TOKEN"
          echo "CODEARTIFACT_TOKEN=$CODEARTIFACT_TOKEN" >> $GITHUB_ENV

      - name: Build package
        run: |
          pip install build
          python -m build

      - name: Publish to CodeArtifact
        env:
          TWINE_USERNAME: aws
          TWINE_PASSWORD: ${{ env.CODEARTIFACT_TOKEN }}
          TWINE_REPOSITORY_URL: https://my-company-123456789012.d.codeartifact.us-east-1.amazonaws.com/pypi/my-repo/
        run: |
          pip install twine
          twine upload dist/*
```

G. Unified Artifact Repository (Artifactory/Nexus)

Repository structure for multiple package types:

```
artifactory.company.com/
├── pypi-local/                 # Python packages
│   ├── simplestat/
│   └── myapp/
├── pypi-remote/                # Proxy to public PyPI
│   └── (cached from pypi.org)
├── pypi-virtual/               # Combined view
│   └── (local + remote)
├── generic-local/              # Tarballs, binaries
│   ├── stats-calc/
│   └── other-projects/
├── docker-local/               # Container images
└── npm-local/                  # Node packages
```

Benefits of unified repositories:

- Single authentication system
- Cross-repository dependencies
- Unified security scanning
- Build provenance tracking
- License compliance management
- Bandwidth optimization (caching)

H. Installing from Private Repositories

Python with authentication:

```
# Using environment variable
export PIP_INDEX_URL=https://user:token@pypi.company.com/simple/
pip install simplestat

# Using pip.conf (Linux/macOS: ~/.pip/pip.conf)
[global]
index-url = https://pypi.company.com/simple/
trusted-host = pypi.company.com

# Using .netrc for credentials (Linux/macOS: ~/.netrc)
machine pypi.company.com
```

```
login myusername
password mytoken
```

Tarballs with authentication:

```
# Using curl with token
curl -H "Authorization: Bearer ${TOKEN}" \
     https://releases.company.com/stats-calc/1.0.0/stats-calc-1.0.0.tar.gz
\
     -o stats-calc-1.0.0.tar.gz

# Using wget with credentials
wget --header="Authorization: Bearer ${TOKEN}" \
     https://releases.company.com/stats-calc/1.0.0/stats-calc-1.0.0.tar.gz
```

I. Security Best Practices

1. Use Tokens, Not Passwords

```
# Store in GitHub Secrets
secrets:
  PYPI_TOKEN: ${{ secrets.PYPI_TOKEN }}
  ARTIFACT_TOKEN: ${{ secrets.ARTIFACT_TOKEN }}
```

2. Scope Permissions Appropriately

- Read-only tokens for installation
- Write tokens only for CI/CD pipelines
- Time-limited tokens when possible

3. Scan Packages for Vulnerabilities

```
- name: Security scan
  run: |
    pip install safety
    safety check --json
```

4. Sign Packages

```
# Sign wheel with GPG
gpg --detach-sign --armor dist/myapp-1.0.0-py3-none-any.whl
```

J. Cost Considerations

GitHub Actions Artifacts:

- Free tier: Limited storage
- Automatic cleanup after retention period
- Good for: CI/CD intermediate files

Private Package Repositories:

- Ongoing storage costs
- Authentication/authorization infrastructure
- Good for: Production releases, long-term storage

Hybrid Approach (Recommended):

- Use GitHub Actions artifacts during build/test
- Promote successful builds to private repository
- Archive old versions to cheaper storage (S3 Glacier, etc.)

K. Learning Exercise: Simulating Private Repository

Exercise: Set up a local PyPI server

```
# Install pypiserver
pip install pypiserver

# Create package directory
mkdir -p ~/pypi-packages

# Copy your wheel there
cp dist/simplestat-1.0.0-py3-none-any.whl ~/pypi-packages/

# Run local PyPI server
pypi-server run -p 8080 ~/pypi-packages
```

Install from your local server:

```
pip install --index-url http://localhost:8080/simple/ simplestat
```

Bonus: Add to GitHub Actions workflow to test installation

```
- name: Start local PyPI server
  run: |
    pip install pypiserver
    mkdir pypi-packages
    cp dist/*.whl pypi-packages/
    pypi-server run -p 8080 pypi-packages &
    sleep 2
- name: Test installation from local PyPI
  run: |
    pip install --index-url http://localhost:8080/simple/ simplestat
    python -c "from simplestat import mean; print(mean([1,2,3]))"
```

VIII. Key Takeaways and Best Practices (3 minutes)

Packaging Best Practices:

- **Always version your packages** using semantic versioning
- **Pin dependencies** to avoid breaking changes
- **Include a README** and proper metadata in pyproject.toml
- **Test your packages** before deploying
- **Use wheels** for Python packages (faster than source distributions)
- **Use tarballs** for source distributions of compiled code
- **Exclude build artifacts** when creating tarballs

GitHub Actions Best Practices:

- **Cache dependencies** to speed up builds
- **Use artifacts** to pass files between jobs
- **Set retention periods** to manage storage costs
- **Name artifacts clearly** with version or commit information
- **Separate concerns**: Build, test, and deploy in different jobs
- **Use matrix builds** for cross-platform applications

Troubleshooting Checklist:

- [ ] Check Python version compatibility
- [ ] Verify all required files are included
- [ ] Ensure pyproject.toml syntax is correct
- [ ] Check artifact upload/download names match
- [ ] Verify job dependencies (needs) are set correctly
- [ ] Review workflow logs for specific error messages
- [ ] For C/C++ projects, ensure build tools are installed
- [ ] Test tarballs extract and build correctly

IX. Final Question (2 minutes)

Which GitHub Actions feature lets you pass files from one job to another?

A. Runners B. Outputs C. Artifacts D. Steps

Answer: C. Artifacts

Explanation: Artifacts allow you to persist data after a job completes and share files between jobs in a workflow. Outputs (B) can pass string data but not files, Runners (A) are the machines that execute jobs, and Steps (D) are individual tasks within a job.

Additional Resources

- Python Packaging User Guide
- GitHub Actions: Using Artifacts
- GitHub Actions: Caching Dependencies
- Semantic Versioning Specification