GitHub Actions

GitHub Actions makes it easy to automate all your software workflows, now with world-class CI/CD. Build, test, and deploy your code right from GitHub. Make code reviews, branch management, and issue triaging work the way you want.

Writing Workflows

GitHub Actions workflows are structured like

```
.github
   | - workflows
   |       | - *.yml
```

Workflows are triggered by events like **on: push, on: pull_request**, etc and runs jobs.

Jobs are tasks in the workflow that runs in a virtual machine (runs-on: ubuntu-latest). These tasks can be run in parallel or sequentially. Jobs run the required steps to complete the workflow

A step is a single sub-task inside a job. It basically runs a script or a command (pre-built action).

An Action is a reusable functionality. These are published by GitHub or the community

Let's look at how to write a simple workflow for performing unit / integration testing, linting and generating reports.
These are simply `.yml` files that reside inside the `workflows` directory under `.github`.
Let's create `ci.yml` inside `workflows`.
Every workflow starts with a name. Let's call ours `CI/CD pipeline` and set the `name` attribute

```yaml
name: CI/CD pipeline
```

Next we set event triggers for the workflow. This is what the workflow will be listening for to start its jobs. We shall listen for push and pull requests on ANY branch.

```yaml
on:
    push:
        branches: ['**']
    pull_request:
        branches: ['**']
```

Great, now we have our workflow that will be triggered by pushing changes to any branch or by opening a pull request. Let's now define the jobs this workflow should run on being triggered along with it's steps. This is the part where a VM will spin up and run the task. We will first have to prepare the VM. This includes installing python and dependencies.

We will name the first job `build` , tell GitHub the OS the runner VM should use and define the expected output. Next we define the steps to take to install and setup the repository and python. Steps are defined by a hyphen. We can describe the steps using the `name` attribute for each step. We use GitHub's pre-built actions to checkout repo, setup python and install dependencies.

```
jobs:
    build:
        runs-on: ubuntu-latest
        outputs:
            python-version: "3.13"
        steps:
            - name: Checkout repository
              uses: actions/checkout@v4

            - name: Set up Python
              uses: actions/setup-python@v4
              id: set-python
              with:
                  python-version: "3.13"

            - name: Cache pip
              uses: actions/cache@v4
              with:
                  path: ~/.cache/pip
                  key: ${{runner.os}}-
pip-${{hashFiles('**requirements.txt')}}
                  restore-keys: | ${{ runner.os }}-pip-

            - name: Install Deps
              run: |
                  python -m pip install --upgrade pip
                  pip install -r requirements.txt
```

We have now prepped the runner with everything it needs to run unit / integration tests. Don't worry about how stuff started getting complex after the 2nd step. Every action has docs on how to use them to help you write workflows.

In short, step 3 is creating a cache for pip so that it doesn't have to re-install dependencies every time. Step 4 is using pip to install dependencies from the `requirements.txt`

Next up, we will write the next job that has to run after `build` . Since we prepped the runner VM, we can now run unit tests or integration tests on the pushed code. Let's create a `test` job that will run after `build` job succeeds. To achieve this, we can use the `needs` attribute.

```
    test:
        runs-on: ubuntu-latest
        needs: build
        steps:
            - name: Checkout Repo
              uses: actions/checkout@v4

            - name: Set up Python
              uses: actions/setup-python@v4
              with:
                  python-version: "3.13"

            - name: Install Deps
              run: |
                  python -m pip install --upgrade pip
                  pip install -r requirements.txt

            - name: Run Tests (verbose)
              run: pytest tests/ -v
```

Alright! We see in the last step of this job, tests will be run using `pytest`. But wait a minute... Why did we re-checkout (fetch the repo into VM), setup python and reinstalled dependencies?

Here is where we need to remember - EVERY job spins up a fresh runner VM. The build job checks if the setup process succeeds. Only if setup succeeds can we move on to testing.

Moving on, let's create a `coverage` job that uploads coverage score of the tests to an html. Of course, to generate the report we would need the tests to be completed.

```
    coverage:
        runs-on: ubuntu-latest
        needs: test
        steps:
            - name: Checkout Repo
              uses: actions/checkout@v4

            - name: Set up Python
              uses: actions/setup-python@v4
              with:
                  python-version: "3.13"
```

```yaml
        - name: Install Deps
          run: |
              python -m pip install --upgrade pip
              pip install -r requirements.txt

        - name: Run Tests with HTML report
          run: pytest tests/ --cov=src --cov-report=html

        - name: Upload coverage HTML report
          uses: actions/upload-artifact@v4
          with:
              name: coverage-html
              path: htmlcov
```

Great, we create an artifact (report file in this case) in `htmlcov` that will be created.

That's mostly it! You now have an understanding on how to write a workflow to automate testing.
Let's complete this workflow with a `lint` job, a `format` job and a `security` job that generates their reports. As with the previous jobs, you can copy the common steps of the jobs but make sure to set up the `needs` attribute appropriately

```yaml
  lint:
    runs-on: ubuntu-latest
    needs: coverage
    steps:
        - # Checkout Repo
        - # Setup Python
        - name: Install pylint
          run: |
              python -m pip install --upgrade pip
              pip install pylint

        - name: Run pylint and save report
          run: |
              pylint src/ --ouput-format=text > lint_report.txt || true

        - name: Upload lint report
          uses: actions/upload-artifact@v4
          with:
              name: lint-report
              path: lint_report.txt

  format:
    runs-on: ubuntu-latest
    needs: lint
    steps:
        - # Checkout Repo
        - # Setup Python
        - name: Install black
          run: |
              python -m pip install --upgrade pip
              pip install black

        - name: Check formatting with black
          run: |
              black src/ tests/ --check

        - name: Show black diff
          run: |
              black src/ test/ --diff || true

security:
    runs-on: ubuntu-latest
    needs: format
    steps:
        - # Checkout Repo
        - # Setup Python
        - name: Install bandit
          run: |
              python -m pip install --upgrade pip
              pip install bandit

        - name: Run bandit (console)
          run: |
              bandit -r src/ -v || true

        - name: Run bandit (json)
          run: |
              bandit -r src/ -f json -o security_report.json || true

        - name: Upload Security Report
          uses: actions/upload-artifact@v4
          with:
              name: security-report
              path: security_report.json
```

That completes the entire workflow. Push this file to your repo, enable the workflow in the "Actions" tab and try pushing changes to watch the workflow execute.

This guide focuses on understanding how to write workflows. Hence, the commands that you see as the value for `run` fields is not part of the guide, but part of the python library / tool itself.