# CLOUD ENGINEERING

**Programming Best Practices**

Ashish Pujari

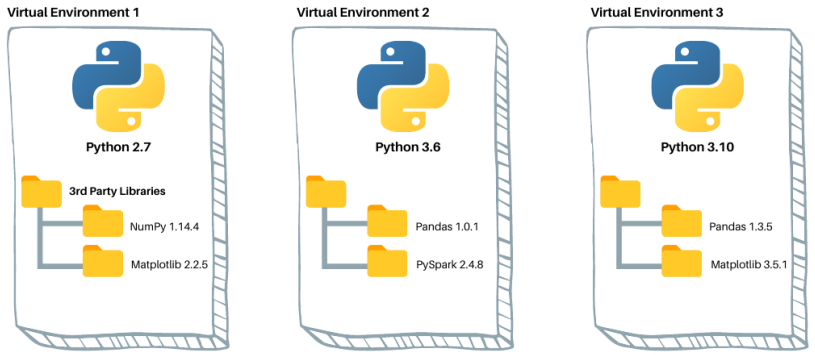
# Lecture Outline

- Environment Management
- Code Management
- Coding Standards
- Logging and Exception Handling
- Unit Testing

# ENVIRONMENT MANAGEMENT

# Virtual Environments

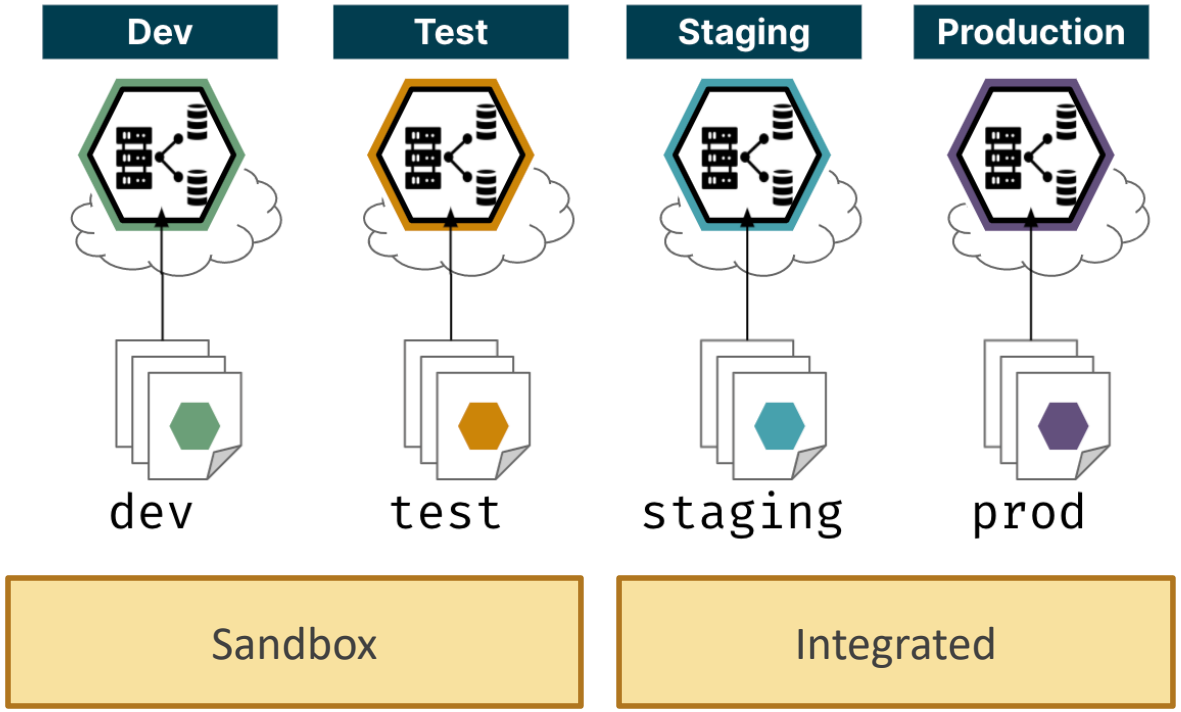
- A Python virtual environment consists of
  - the Python interpreter that the virtual environment runs on
  - a folder containing third-party libraries installed in the virtual environment.




dataquest.io

- Best Practices
  - Use a Separate Virtual Environment for Each Project
  - Don’t Forget to Activate Your Python Virtual Environment
  - Don’t Use >= for Package Versioning in a Python Virtual Environment

# ML/AI Environments



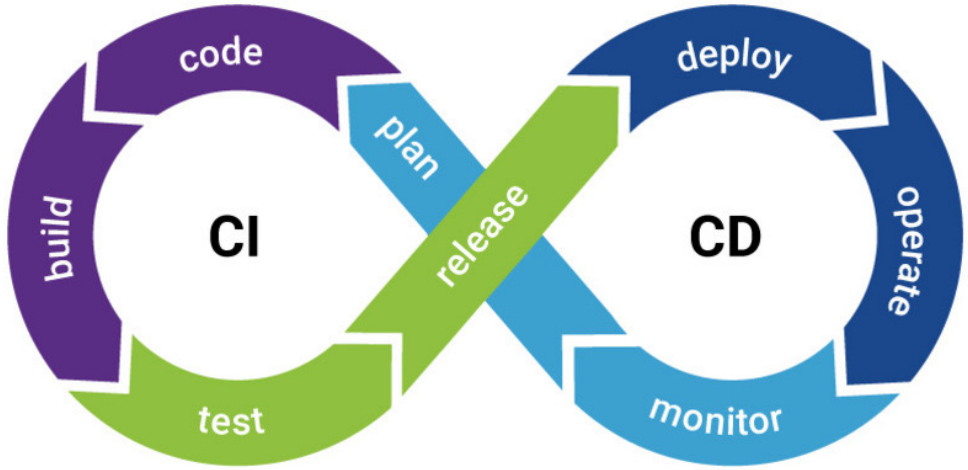
# Configuration Variables

- Set up different configuration options for different programming environments
  - E.g., API keys, connection strings, logging configuration, or debug options, or other configuration settings
- Methods to store configuration:
  - Built-in data structures
  - Using dynamic loading
  - Environment variables - global variables; should be minimal or avoided
  - External Configuration Files -.ini, .yaml, .json, .xml, etc.

# Continuous Integration/Delivery/Deployment (CI/CD)

- Modern software development practice in which incremental code changes are made frequently and reliably
- Streamlined workflows through built-in automation, testing
- Advantages:
  - Accelerated time-to-value
  - Increased visibility
  - Frequent iterations
  - Increases efficiency
  - Higher team collaboration
  - Stable testing environments

# CI/CD: Tools




GitLab CI

AWS CodePipeline

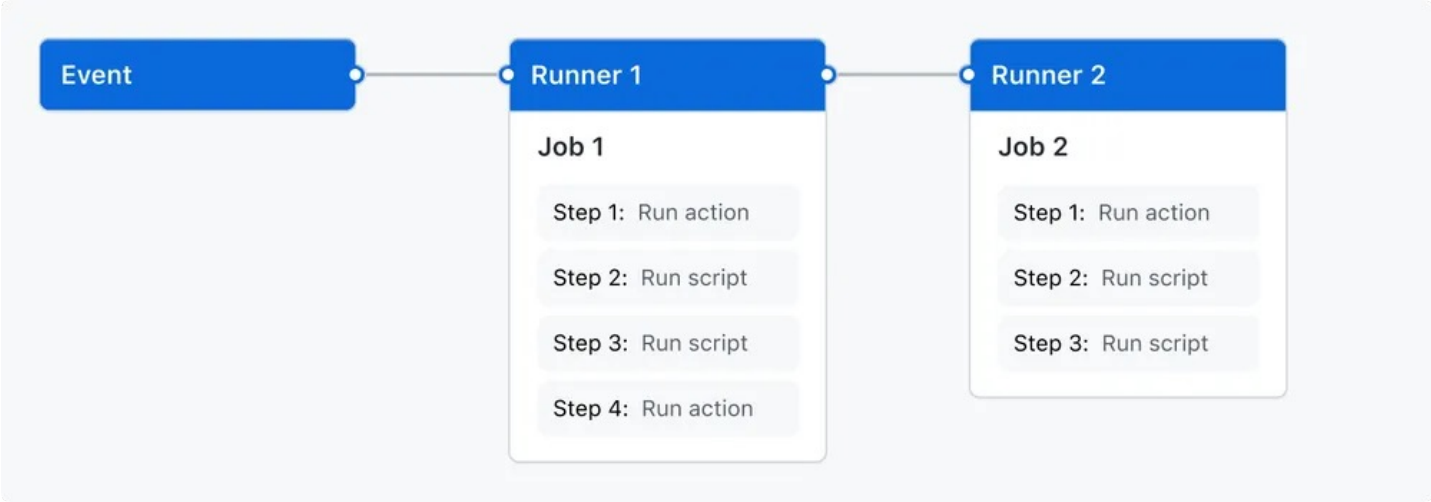
Jenkins

GitHub Actions

Azure Pipelines

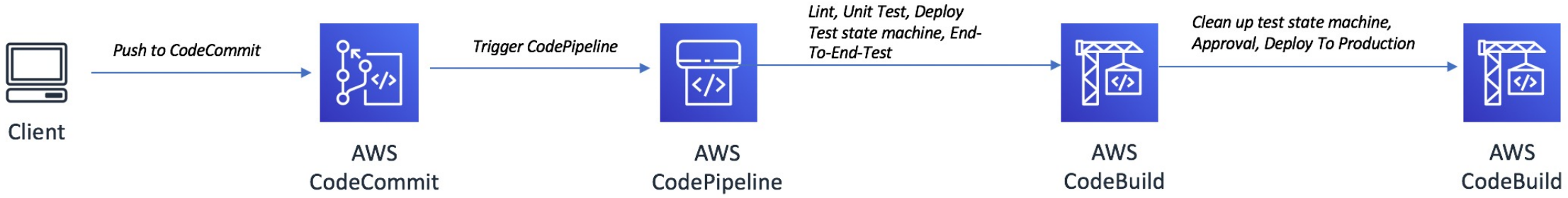
# CI/CD: Github

- GitHub Actions makes it easy to automate software workflows
- Use cases: CI/CD, Static/Dynamic analysis, Code building, deployment, etc.




- Further reading
  - https://docs.github.com/en/actions
  - https://github.com/sdras/awesome-actions#machine-learning-ops

# CI/CD: AWS CodePipeline



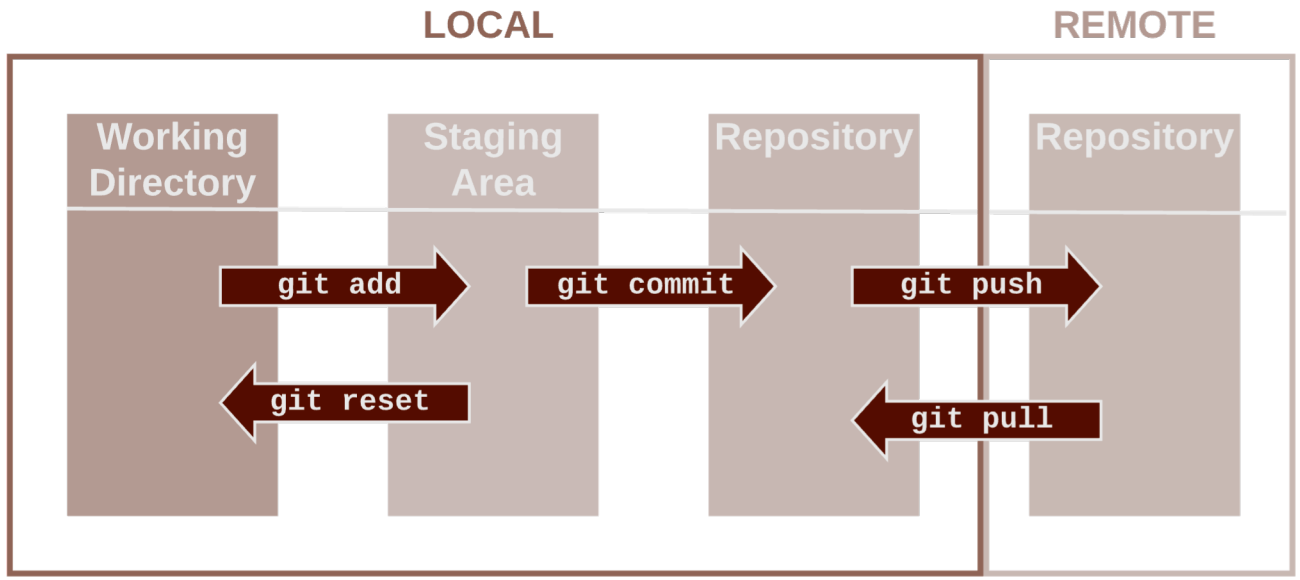

- Further reading
  - AWS CodePipeline

# CODE MANAGEMENT

Versioning, Git

# Code Versioning

- Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later
- Git is a free and open-source distributed version control system




https://git-scm.com/book/en/v2/Getting-Started-What-is-Git

# Git Commands

## Create and Config

```
$ git init
```
Turn an existing directory into a git repository

```
$ git clone [url]
```
Clone (download) a repository that already exists on GitHub, including all of the files, branches, and commits

```
$ git config --global user.name "[name]"
```
Sets the name you want attached to your commit transactions

```
$ git config --global user.email "[email address]"
```
Sets the email you want attached to your commit transactions

```
$ git config --global color.ui auto
```
Enables helpful colorization of command line output

## Make Changes

```
$ git log
```
Lists version history for the current branch

```
$ git log --follow [file]
```
Lists version history for a file, including renames

```
$ git diff [first-branch]...[second-branch]
```
Shows content differences between two branches

```
$ git show [commit]
```
Outputs metadata and content changes of the specified commit

```
$ git add [file]
```
Snapshots the file in preparation for versioning

```
$ git commit -m "[descriptive message]"
```
Records file snapshots permanently in version history

# Git Commands

## Synchronize Changes

**`$ git fetch`**
Downloads all history from the remote tracking branches

**`$ git merge`**
Combines remote tracking branch into current local branch

**`$ git push`**
Uploads all local branch commits to GitHub

**`$ git pull`**
Updates your current local working branch with all new commits from the corresponding remote branch on GitHub. `git pull` is a combination of `git fetch` and `git merge`

## Branches

**`$ git branch [branch-name]`**
Creates a new branch

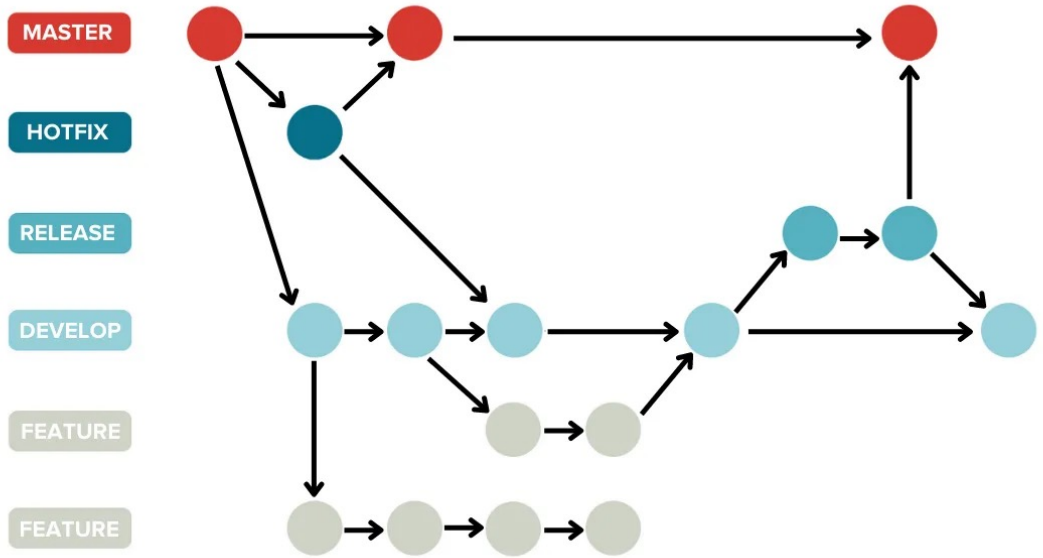
**`$ git checkout [branch-name]`**
Switches to the specified branch and updates the working directory

**`$ git merge [branch]`**
Combines the specified branch's history into the current branch. This is usually done in pull requests, but is an important Git operation.

**`$ git branch -d [branch-name]`**
Deletes the specified branch

# Git Branching

- Branches are used as a means for teams to develop features giving them a separate workspace for their code.
- These branches are usually merged back to a main branch upon completion of work.

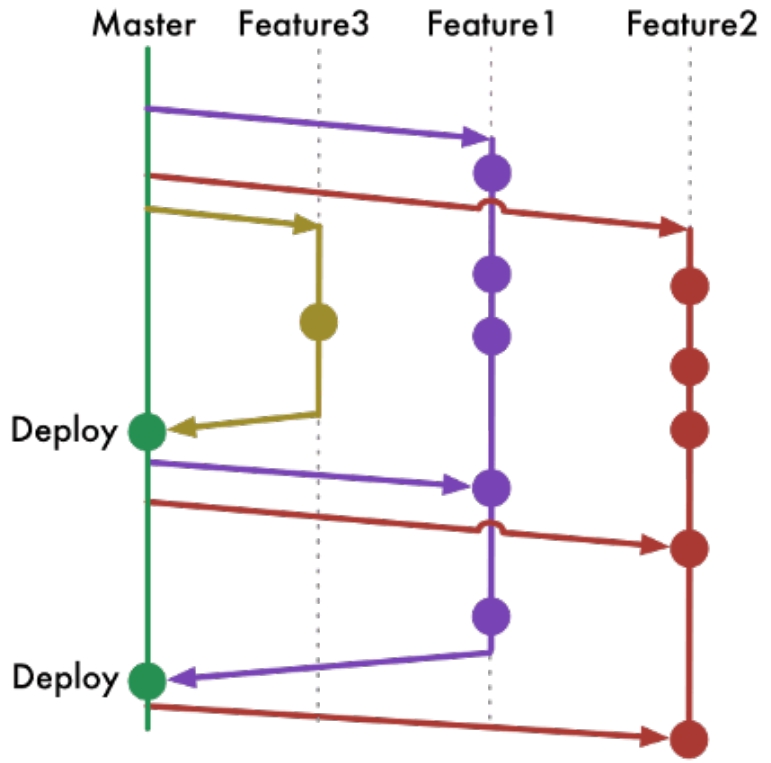


# Common Types of Branches

| Branch Type | Description |
|---|---|
| Trunk branch | Main, mainline, or the master branch; Implicit first branch when a repo is created |
| Development branch | Long-lived feature branch that holds changes made by developers before they're ready to go to production; Often parallels the main or trunk branch |
| Feature branch | Used for the lifetime of a new feature during its development<br>Often used by a single developer, but possible to share it with others |
| Release branch | Reflects a set of changes that are intended to go through the production release process |
| Hotfix branch | changes related to emergency bug fixes; used in teams with explicitly versioned products, such as installed applications |

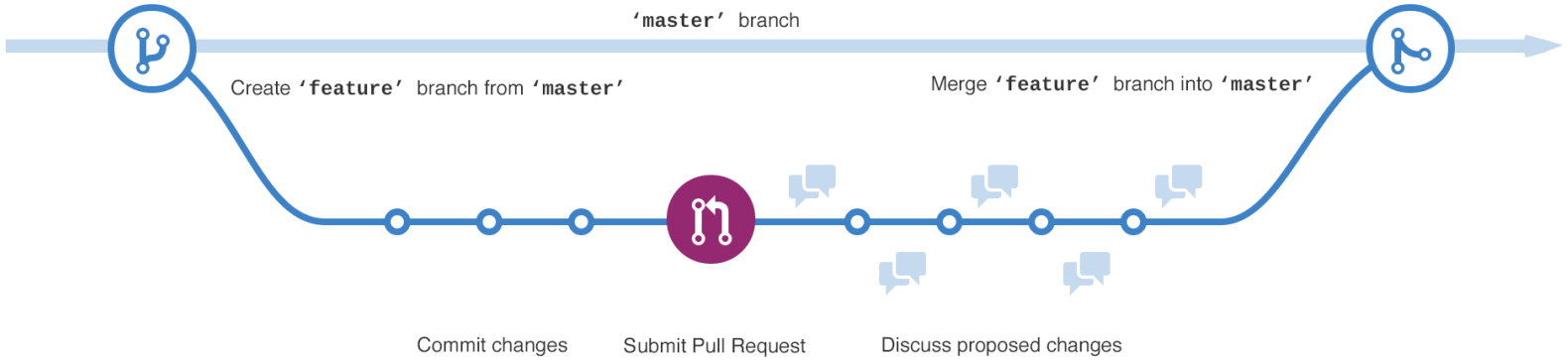
# Branching Strategies

- Designed to enable parallel development and structure releases by avoiding merge conflicts and allow for the easier integration of new features
- E.g., GitFlow, GitLab Flow, Trunk development



# Pull Request (PR)

- Step 1: Push your local branch to the remote Git repository.
- Step 2: Write a description in your pull requests that explain:
  - What problem(s) are being solved
  - The approach and changes made to address it
  - Any special notes or highlights in the code review.
- Step 3: Your peer will review the code and will merge it with main branch



# Git: Best Practices

- Frequent Commits
  - Commit frequently, keeping each commit minimal but complete.
- Large Files Management
  - Store large files, data , logs, models, etc. externally to avoid bloating the repository size.
- Structured Commit Messages
  - Format commit messages with a header, body, and footer.
- Branch Management
  - Keep branch names concise and purposeful, ensuring branches have a single clear objective.
- Rebasing Before Merging
  - Before merging feature branches into public branches (e.g., main), rebase your code onto the latest version of the public branch and resolve conflicts locally.

# CODING STANDARDS

# Programing Paradigms

1. Imperative Programming
2. Object-Oriented Programming (OOP)
3. Functional Programming
4. Procedural Programming
5. Event-Driven Programming
6. Logic Programming
7. Aspect-Oriented Programming (AOP)
8. Concurrent Programming
9. Symbolic Programming
10. Natural Language Programming

# Programming Best Practices

- DRY Principle – Don’t repeat yourself
- Keep code simple and readable
- Understand context
- Modular design
- Low coupling and high cohesion
- Avoid Deep Nesting
- Naming conventions, file and folder structure
- Follow coding standards
- Proper logging and exception handling
- Peer Review
- Unit Testing
- Code coverage
- Code pipeline automation

# Coding Standards

- Rules, techniques, and guidelines to create cleaner, better readable, and more efficient code with minimal bugs and errors
- E.g., PEP-8 is a set of guidelines for writing Python code that is easy to read and maintain.
- Further Reading:
  - https://effectivepython.com/
  - https://datasciencecampus.github.io/coding-standards/python.html#style-guide

# PEP-8: Best Practices

| | |
|---|---|
| Naming Conventions | • Use a consistent naming convention for all variables, functions, and classes.<br>• Variable names should be lowercase, with words separated by underscores (snake_case), while class names should be in CamelCase.<br>• Function names should also be lowercase, with words separated by underscores, and should be descriptive of their purpose. |
| Indentation | • Use 4 spaces for indentation, instead of tabs. This helps ensure that code is readable across different platforms and text editors. |
| Line Length | • Keep lines of code to a maximum of 79 characters in length, to make it easier to read and understand code. This can be extended to 120 characters if necessary. |
| Whitespace | • Use whitespace judiciously to make code more readable.<br>• Separate functions and classes with two blank lines, and code blocks within functions and classes with a single blank line.<br>• Use a single space around operators and after commas in function calls. |

# PEP-8: Best Practices

| | |
|---|---|
| Comments | • Use comments sparingly, and only when they add value to the code.<br>• Use comments to explain why code is being done, not what it is doing, and to use complete sentences with proper grammar and spelling. |
| Import Statements | • Place all import statements at the top of the file, and group them in the following order: standard (built-in) library imports, third-party library imports, local application imports.<br>• Use absolute imports, rather than relative imports (dot notation) |
| Function and Method Arguments | • Use whitespace around the equals sign when defining function and method arguments with default values.<br>• Use a space before and after the equals sign |

# Code Analysis

- Static code analysis
  - Examines code to identify issues within the logic and techniques.
  - Linting is static analysis process used to flag patterns that might cause errors or other problems
  - Tools: PyLint, pyflakes, Mypy, ast, etc.

- Dynamic code analysis
  - Running code and examining the outcome
  - Performs testing possible execution paths of the code
  - Tools: DynaPyt, pdb, etc.

# AI Coding Assistants

- Advantages:
  - Code completion
  - Refactoring support
  - Error and vulnerabilities detection
  - Enforces coding standards
  - Language Agnostic
  - Review Assistance
  - Documentation Generation


Google Cloud Code

GitHub Copilot

# LOGGING AND EXCEPTION HANDLING

# Logging

- Logging is used to monitor system/model performance, troubleshooting errors, root cause analysis, cyber security incidents, etc.
- Typical log file contains:
  - Date and time when event occurred
  - Log level or severity level
  - An error code, if applicable
  - User id or process id that triggered the event
  - Message: description of the event
  - Function name
  - Name or IP address of the device where the event took place

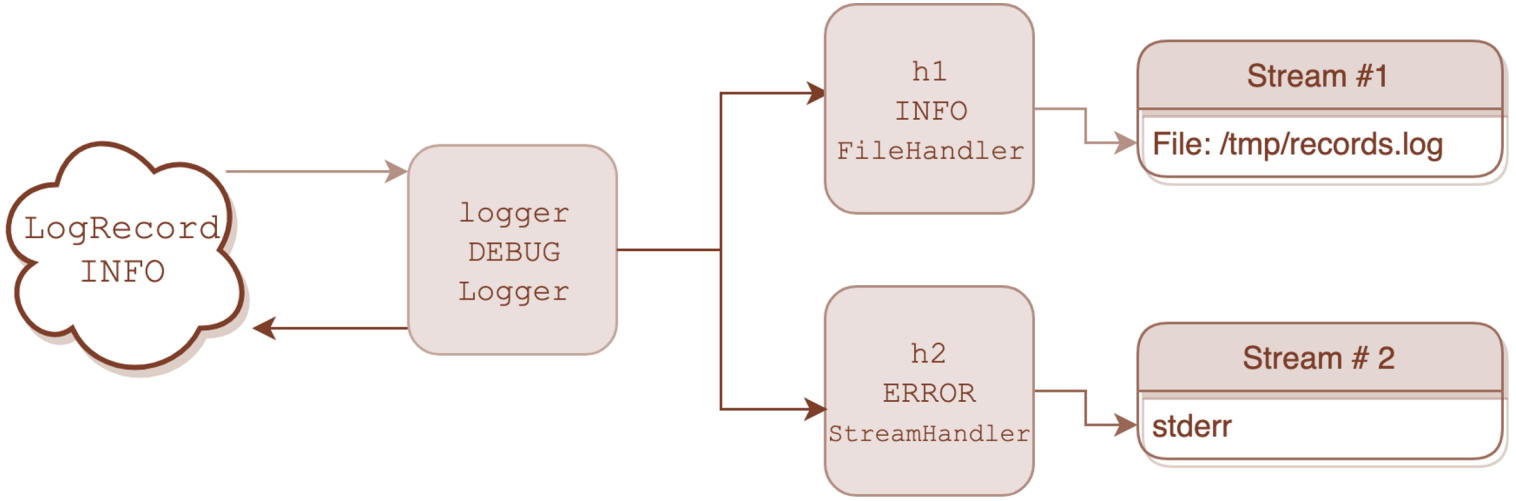
# Log Levels

- Describe the type and severity of a logged event based on the severity of the impact on users and the urgency of response required by the organization.

| Log Level | Details | What to log | Environment |
|---|---|---|---|
| DEBUG | Detailed information, for diagnosis and troubleshooting. | Input and intermediate values, time taken for major/minor subroutines, etc. | Dev, Test |
| INFO | High level logs with small amount of information; Confirmation that things are working as expected. | Major branches in processing logic, time taken for major subroutines, size of input/output, record counts, etc. | Dev, Test, Prod |
| WARNING | Logs unexpected events and potential future problems. | Unused values/data, library or version deprecation, etc. | Dev, Test, Prod |
| ERROR | Serious problems; code has not been able to perform certain functions. | Unexpected input or parameter values, file not found, divide by zero, etc. | Dev, Test, Prod |
| CRITICAL | Serious errors; the program execution may have failed. | Resource Leaks, Data loss, Disk full, etc. | Dev, Test, Prod |

# Log Handlers

- Object that handles how and where the logs must be directed
  - StreamHandler ,FileHandler, RotatingFileHandler, etc.




https://realpython.com/python-logging-source-code/

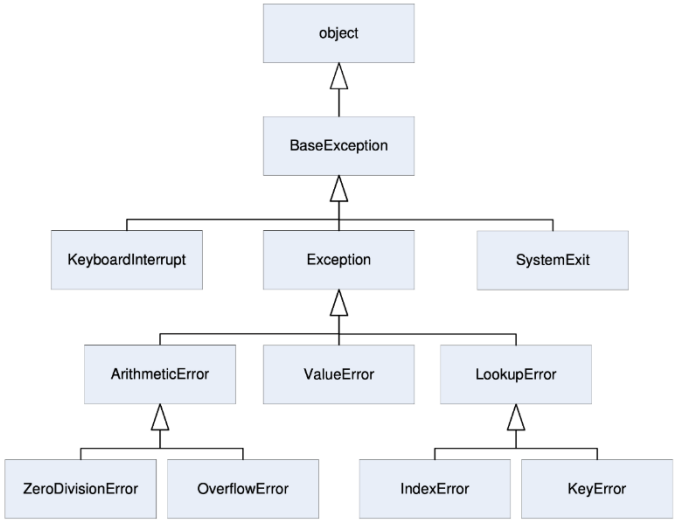
# Exceptions

- Errors detected during execution (run-time) are called exceptions
- Some exceptions are built in; sometimes useful to create user-defined exceptions
- It is possible to write programs that handle selected exceptions.

Context where the exception occurred

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

Exception details




https://docs.python.org/3/tutorial/errors.html

# Handling Exceptions

A *try* may have more than one *except* clause, to specify handlers for different exceptions

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("OS error:", err)
except ValueError:
    print("Could not convert data to an integer.")
except Exception as err:
    print(f"Unexpected {err=}, {type(err)=}")
    raise
```

Base class of all non-fatal exceptions

Cleanup action

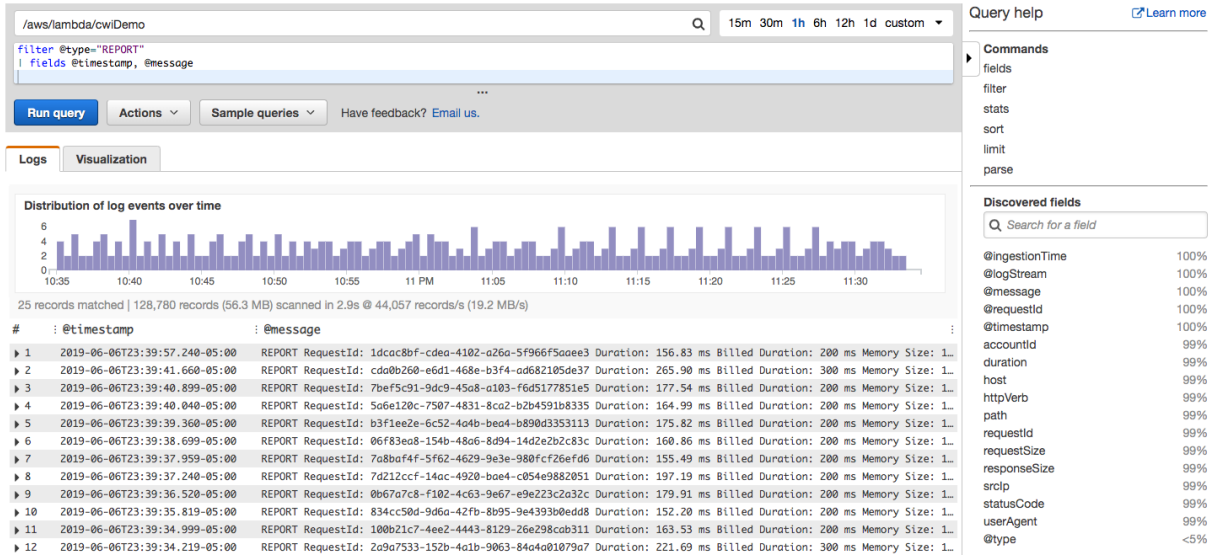
```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print('Goodbye, world!')
...
Goodbye, world!
KeyboardInterrupt
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
```

# Handling Exceptions: Best Practices

- Use try-except-finally blocks to handle exceptions gracefully and ensure your code can handle errors effectively
- Explicit exception handling is better than implicit
- Use a finally block to perform cleanup operations such as releasing resources e.g., database connection, threads, etc.
- Keep code in your try block to a minimum and avoid handling too many exceptions
- Use custom exceptions only when absolutely necessary

# Monitoring


Sent/Deleted Messages
Count
29.4k
14.7k
0
17:00 18:00 19:00 20:00 21:00 22:00 23:00 00:00 01:00 02:00 03:00 04:00 05:00 06:00 07:00 08:00 09:00 10:00 11:00 12:00 13:00 14:00 15:00 16:00
NumberOfMessagesSent
NumberOfMessagesDeleted
CPU
Percent
90
61.3
32.5
19:00 22:00 01:00 04:00 07:00 10:00 13:00 16:00
CPUUtilization
Network
Bytes
320M
217M
114M
19:00 22:00 01:00 04:00 07:00 10:00 13:00 16:00
NetworkIn
NetworkOut


/aws/lambda/cwiDemo
15m 30m 1h 6h 12h 1d custom
filter @type="REPORT"
| fields @timestamp, @message
Run query
Actions
Sample queries
Have feedback? Email us.
Logs
Visualization
Distribution of log events over time
6
4
2
0
10:35 10:40 10:45 10:50 10:55 11 PM 11:05 11:10 11:15 11:20 11:25 11:30
25 records matched | 128,780 records (56.3 MB) scanned in 2.9s @ 44,057 records/s (19.2 MB/s)
# @timestamp @message
1 2019-06-06T23:39:57.240-05:00 REPORT RequestId: 1dcac8bf-cdea-4102-a26a-5f966f5aaee3 Duration: 156.83 ms Billed Duration: 200 ms Memory Size: 1…
2 2019-06-06T23:39:41.660-05:00 REPORT RequestId: cda0b260-e6d1-468e-b3f4-ad682105de37 Duration: 265.90 ms Billed Duration: 300 ms Memory Size: 1…
3 2019-06-06T23:39:40.899-05:00 REPORT RequestId: 7bef5c91-9dc9-45a8-a103-f6d5177851e5 Duration: 177.54 ms Billed Duration: 200 ms Memory Size: 1…
4 2019-06-06T23:39:40.040-05:00 REPORT RequestId: 5a6e120c-7507-4831-8ca2-b2b4591b8335 Duration: 164.99 ms Billed Duration: 200 ms Memory Size: 1…
5 2019-06-06T23:39:39.360-05:00 REPORT RequestId: b3f1ee2e-6c52-4a4b-bea4-b890d3353113 Duration: 175.82 ms Billed Duration: 200 ms Memory Size: 1…
6 2019-06-06T23:39:38.699-05:00 REPORT RequestId: 06f83ea8-154b-48a6-8d94-14d2e2b2c83c Duration: 160.86 ms Billed Duration: 200 ms Memory Size: 1…
7 2019-06-06T23:39:37.959-05:00 REPORT RequestId: 7a8baf4f-5f62-4629-9e3e-980fcf26efd6 Duration: 155.49 ms Billed Duration: 200 ms Memory Size: 1…
8 2019-06-06T23:39:37.240-05:00 REPORT RequestId: 7d212ccf-14ac-4920-bae4-c054e9882051 Duration: 197.19 ms Billed Duration: 200 ms Memory Size: 1…
9 2019-06-06T23:39:36.520-05:00 REPORT RequestId: 0b67a7c8-f102-4c63-9e67-e9e223c2a32c Duration: 179.91 ms Billed Duration: 200 ms Memory Size: 1…
10 2019-06-06T23:39:35.819-05:00 REPORT RequestId: 834cc50d-9d6a-42fb-8b95-9e4393b0edd8 Duration: 152.20 ms Billed Duration: 200 ms Memory Size: 1…
11 2019-06-06T23:39:34.999-05:00 REPORT RequestId: 100b21c7-4ee2-4443-8129-26e298cab311 Duration: 163.53 ms Billed Duration: 200 ms Memory Size: 1…
12 2019-06-06T23:39:34.219-05:00 REPORT RequestId: 2a9a7533-152b-4a1b-9063-84a4a01079a7 Duration: 221.69 ms Billed Duration: 300 ms Memory Size: 1…
Query help
Learn more
Commands
fields
filter
stats
sort
limit
parse
Discovered fields
Search for a field
@ingestionTime 100%
@logStream 100%
@message 100%
@requestId 100%
@timestamp 100%
accountId 99%
duration 99%
host 99%
httpVerb 99%
path 99%
requestId 99%
requestSize 99%
responseSize 99%
srcIp 99%
statusCode 99%
userAgent 99%
@type <5%

# UNIT TESTING

# Unit Testing

- Unit testing is essential for early bug detection and fixing, simplifying integration and minimizing code regression
- Characteristics
  - Fast
  - Isolated
  - Repeatable
  - Reliable
  - Named properly


# Benefits of Unit Testing

- Testing outputs
  - Check if a piece of code (component, function, method, or class) handling a logic gives the expected output.
- Reduce development costs as the application scales
  - Discover and eliminate bugs at the early development stage.
  - Evaluate the effects of recent modifications on other code units.
  - Increase software validity, integrity, and quality, as other programmers or end-users trust a tested product better.
- Documentation tool
  - Helps developers understand the working mechanism of the code or component

# Test Planning

Create a test plan before writing tests that covers :

1. What your code does
2. Good and bad case scenarios
3. What does not require testing
4. The unit tests themselves

# Patterns

- Arrange, Act and Assert pattern to organize unit tests.
  - Arrange phase, all the objects and variables needed for the test are set.
  - Act phase, the function/method/class under test is called.
  - Assert phase, we verify the outcome of the test.
- Mocking Data
  - Allows us to make calls to external resources without actually invoking them.
  - You can call functions, classes and objects and determine what values, if any, those resources return.
- Parameterization
  - Run the same unit test several times with multiple parameters

# Unit Testing: Machine Learning

| Unit Tests | Description |
|---|---|
| Data preprocessing | Verify the correctness of data preprocessing steps such as data normalization, scaling, one-hot encoding, or any other data transformation |
| Model architecture | Verify the structure and configuration of the model. E.g., expected number of layers, the correct activation functions, appropriate input and output shapes |
| Model training | Evaluate the model training process. E.g., if the model converges during training, if the loss decreases over epochs, and if the model's performance improves. |
| Model evaluation | Assess the performance of the trained model on an unseen dataset. E.g., accuracy, precision, recall, F1 score, etc. |
| Integration | Checking the integration of various components of a machine learning system |
| Data leakage | Examine whether the model is inadvertently using information from the test set during training, which can lead to overly optimistic performance metrics. E.g., Target leakage, Train-test contamination, Data preprocessing leakage |
| Robustness | Evaluate the model's performance under various scenarios, such as handling missing values, outliers, or different distributions of the input data |
| Deployment | Validate the functionality and performance of the deployed model in its intended (prod/non-prod) environment. |

# Code Coverage

- Code coverage is a measure of exhaustiveness of a test suite
- Higher the code coverage the fewer defects a system tends to have
- Types of coverage
  - Branch Coverage: validates whether a test covers the code execution path
  - Statement Coverage: validates the statements or actions within a code path
  - Decision Coverage: subset of branch coverage that verifies if a test covers all the conditions
  - Loop Coverage: validates the number (%) of loops that run at least once under a test suite
  - Function Coverage: percentage of code functions that run during testing