Module 3: Software Development and Design

Module Objectives

- Module Title: Software Development and Design
- Module Objective: Use software development and design best practices.
- It will comprise of the following sections:

Topic Title	Topic Objective
3.1 Software Development	Compare software development methodologies.
3.2 Software Design Patterns	Describe the benefits of various software design patterns.
3.3 Version Control Systems	Implement software version control using GIT.
3.4 Coding Basics	Explain coding best practices.
3.5 Code Review and Testing	Use Python Unit Test to evaluate code.
3.6 Understanding Data Formats	Use Python to parse different messaging and data formats.

3.1 Software Development

Introduction

- The software development process is also known as the software development life cycle (SDLC).
- SDLC is more than just coding and also includes gathering requirements, creating a proof of concept, testing, and fixing bugs.

Software Development Life Cycle (SDLC)

- SDLC is the process of developing software, starting from an idea and ending with delivery. This process consists of six phases. Each phase takes input from the results of the previous phase.
- SDLC is the process of developing software, starting from an idea and ending with delivery. This process consists of six phases. Each phase takes input from the results of the previous phase.
- Although the waterfall methods is still widely used today, it's gradually being superseded by more adaptive, flexible methods that produce better software, faster, with less pain. These methods are collectively known as "Agile development."

Requirements and Analysis Phase

- **The requirements and analysis phase** involves the product owner and qualified team members exploring the stakeholders' current situation, needs and constraints, present infrastructure, and so on, and determining the problem to be solved by the software.
- After gathering the requirements, the team analyzes the results to determine the following:
  - Is it possible to develop the software according to these requirements, and can it be done on-budget?
  - Are there any risks to the development schedule, and if so, what are they?
  - How will the software be tested?
  - When and how will the software be delivered?
- At the conclusion of this phase, the classic waterfall method suggests creating a Software Requirement Specification (SRS) document, which states the software requirements and scope, and confirms this meticulously with stakeholders.

Design and Implementation Phases

- **Design**
  - During the Design phase, the software architects and developers design the software based on the provided SRS.
  - At the end of the phase, the team creates High-Level Design (HLD) and Low-Level Design (LLD) documents.
- **Implementation**
  - The implementation phase is also called the coding or development phase.
  - it is the longest phase of the life cycle.
  - As all the components and modules are built during this phase according to low-level and high-level design documents.
  - At the end of the phase, the functional code that implements all customer's requirements is ready to be tested.

Testing, Deployment, and Maintenance Phases

- **Testing**
  - In this phase, code is installed in the test environment
  - Functional testing, integration testing, performance testing and security testing is performed.
  - Testing continues until all the codes are bug free and pass all the tests. At the end of this phase, a high quality, bug-free, working piece of software is ready for production.
- **Deployment**
  - During this phase, the software is installed into the production environment.
  - At the end of the phase, the product manager releases the final piece of software to end users.

Testing, Deployment, and Maintenance Phases

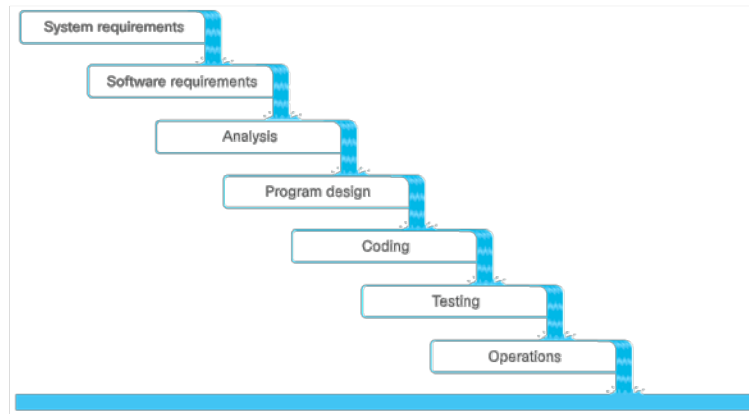
- **Maintenance**
  - During the maintenance phase, the team:
    - Provides support to customers
    - Fixes bugs found in production
    - Works on software improvements
    - Gathers new requests from the customer
  - At the end, the team works on the next iteration and version of the software.

Software Development Methodologies

- A software development methodology is also known as Software Development Life Cycle model.
- The three most popular methodologies are:
  - Waterfall
  - Agile
  - Lean
- The type of methodology to be used depends on the:
  - Type of the project
  - Length of the project
  - Size of the team.

Waterfall Software Development

- The original **waterfall model** was created by Winston W. Royce.
- His original model consisted of seven phases:
  - System requirements
  - Software requirements
  - Analysis
  - Program design
  - Coding
  - Testing
  - Operations
- Note: Each phase cannot overlap and must be completed before the next step starts.



Agile Software Development

- Agile method is flexible and customer-focused.
- A group of 17 software developers came up with the Manifesto for Agile Software Development, also known as the Agile Manifesto, in 2001. According to the Agile Manifesto, the values of Agile are:
  - Individuals and interactions over processes and tools
  - Working software over comprehensive documentation
  - Customer collaboration over contract negotiation

Agile Manifesto Principles			
Customer focus	Collaboration	Working software	Simplicity
Embrace change and adapt	Motivated teams	Work at a sustainable pace	Self-organizing teams
Frequent delivery of working software	Face-to-face conversations	Agile environment	Continuous Improvement

Agile Methods

- The popular Agile methods are:
  - **Agile Scrum**: The Scrum focuses on small, self-organizing teams that meet daily for short periods and work in iterative sprints.
  - **Lean**: The Lean method emphasizes on elimination of wasted effort in planning and execution, and reduction of programmer cognitive load.
  - **Extreme Programming (XP)**: XP deliberately addresses the specific kinds of quality-of-life issues faced by the software development teams.
  - **Feature-Driven Development (FDD)**: FDD prescribes that software development should proceed in terms of an overall model, broken out, planned, designed, and built feature-by-feature.

Agile Methods

- **Sprints**
  - A sprint is a specific period of time, usually between 2-4 weeks, during which, each team takes on as many tasks (also known as user stories) as they feel they can accomplish. When the sprint is over, the software should be working and deliverable.
  - The duration of the sprint is determined before the process begins and should rarely change.
- **Backlog**
  - The backlog consists of all the features of the software, in a prioritized list.

Agile Methods

- **User stories**
  - A user story is a simple statement of what a user (or a role) needs, and why. Each user story should be small enough that a single team can finish it within a single sprint.
  - The suggested template for a user story is:

As a `<user|role>`, I would like to `<action>`, so that `<value|benefit>`

Agile Methods

- **Scrum Teams**
  - Scrum teams are cross-functional, collaborative, self-managed and self-empowered.
  - The scrum teams should not be larger than 10 individuals.
  - The scrum master should have a daily stand-up meeting with the team at a fixed time everyday for not more than 15 minutes.
  - The goal is to go over important tasks that have been finished, are in progress, or are about to be started.

Lean Software Development

- Lean software development is based on Lean Manufacturing principles, which are focused on minimizing waste and maximizing value to the customer.
- The customer determines the useful value of software product features.
- The seven principles of lean, given in the book “Lean Software Development: An Agile Toolkit,” are as follows:
  - Eliminate waste
  - Amplify learning
  - Decide as late as possible
  - Deliver as fast as possible
  - Empower the team
  - Build integrity in
  - Optimize the whole

Lean Software Development

- **Eliminate waste**
  - It is the most fundamental lean principle.
  - There are seven wastes of software development:
    - Partially done work
    - Extra processes
    - Extra features
    - Task switching
  - Waiting
  - Motion
  - Defects

Lean Software Development

- **Amplify Learning with Short Sprints**
  - To be able to fine tune a software, there should be frequent short iterations of working software. This enables the following:
- Developers learn faster
  - Customers can give feedback sooner
  - Features can be adjusted so that they bring customers more value
- **Decide as Late as Possible**
  - When there is uncertainty, it is best to delay the decision-making until as late as possible in the process. This is because it is better to base decisions on facts rather than opinions or speculations.

Lean Software Development

- **Deliver as Fast as Possible**
  - Enables customers to provide feedback
  - Enables developers to amplify learning
  - Provides customers the required features
  - Doesn't allow customers to change their mind
  - Makes everyone take decisions faster
  - Produces less waste

Lean Software Development

- **Empower the Team**
  - Each person must be allowed to make decisions in the area of their own expertise.
- **Build Integrity In**
  - Integrity for the software is when the software addresses the customer’s needs as well as maintains the usefulness for the customer.
- **Optimize the Whole**
  - The software must be built cohesively. The value of the software will suffer if each expert only focuses on their expertise and doesn't consider the ramifications of their decisions on the rest of the software.

Explore Python Development Tools

- Enter the following command to use the **venv** tool to create a Python 3 virtual environment with the name **devfund**. The **-m** switch tells Python to run the **venv** module.

```
python3 -m venv devfund
```

- Run the **pip3 freeze** command to output a list of installed Python packages and verify that there are no additional Python packages currently installed in the environment.

```
pip3 freeze
```

- Note: We will discuss data formats in more detail later.

3.2 Software Design Patterns

Introduction

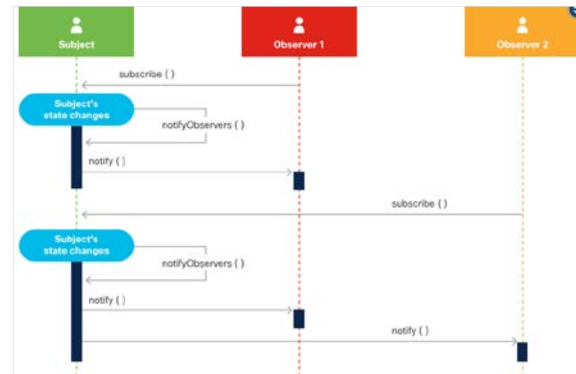
- Software design patterns are best practice solutions for solving common problems in software development.
- Design patterns are language-independent.
- In 1994, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (known as the Gang of Four (GoF)) published a book called Design Patterns - Elements of Reusable Object-Oriented Software. Patterns identified are:
  - Program to an interface, not an implementation.
  - Favor object composition over class inheritance.
- Software design patterns have already been proven to be successful, so using them can speed up development.

The Original Design Patterns

- The Gang of Four divided patterns into three main categories:
  - Creational
  - Structural
  - Behavioral
- They listed 23 design patterns.
- Two of the most commonly used design patterns are:
  - The Observer design pattern (a Behavioral design pattern)
  - The Model-View-Controller (MVC)

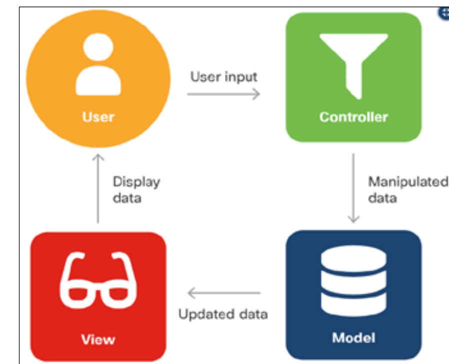
Observer Design Pattern

- The observer design pattern is a subscription notification design that lets objects receive events when there are changes to an object they are observing.
- To implement this subscription mechanism:
  - The subject must have the ability to store a list of all of its observers.
  - The subject must have methods to add and remove observers.
- The benefit of the observer design pattern is that observers can get real time data from the subject when a change occurs.
- Subscription mechanisms always provide better performance than other options, such as polling.



Model-View-Controller (MVC)

- The Model-View-Controller (MVC) design pattern aims to simplify development of applications that depend on graphic user interfaces.
- MVC abstracts code and responsibility into three different components:
  - **Model**: The model is the application's data structure and is responsible for managing the data, logic and rules of the application. It gets input from the controller.
  - **View**: It accepts selected data and displays the visual representation to the user.
  - **Controller**: The controller is the middleman between the model and view. It takes in user input and manipulates it to fit the format for the model or view.
- The benefit of MVC is that each component can be built in parallel.



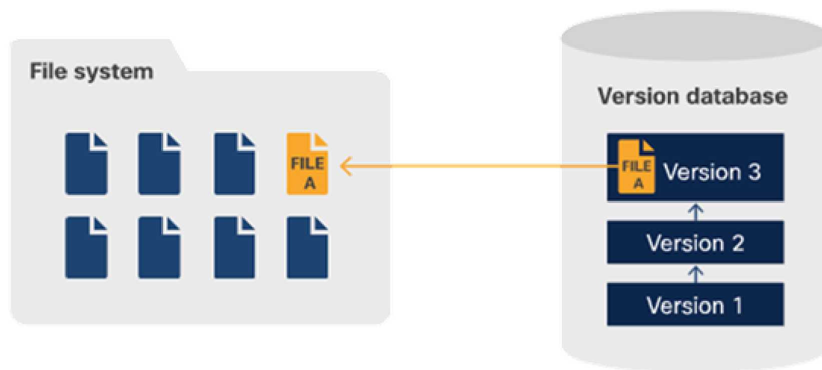
3.3 Version Control Systems

Types of Version Control Systems

- Version control, also called version control systems, revision control or source control, is a way to manage changes to a set of files in order to keep a history of those changes.
- Benefits of version control are:
  - Enables collaboration
  - Accountability and visibility
  - Work in isolation
  - Safety
  - Work anywhere
- There are three types of version control systems:
  - Local
  - Centralized
  - Distributed

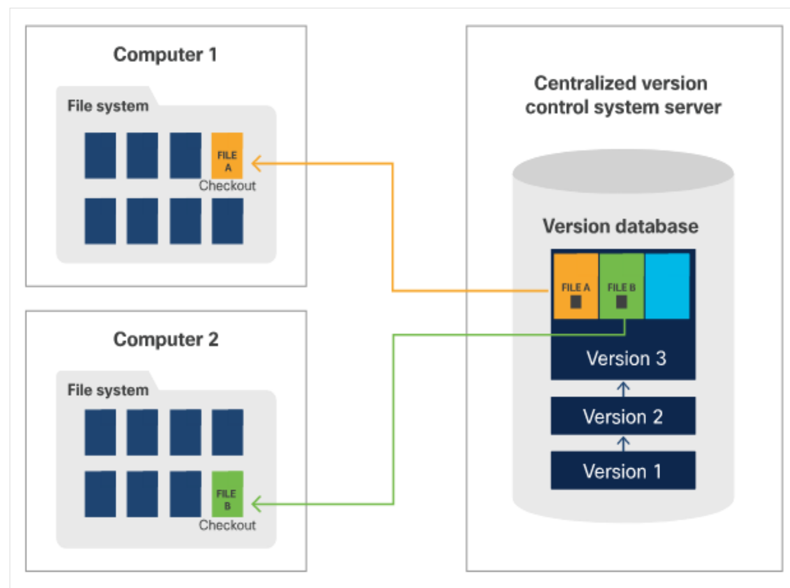
Types of Version Control Systems

- **Local Version Control System (LVCS)**
  - LVCS uses a simple database to keep track of all of the changes to the file.
  - In most cases, the system stores the delta between the two versions of the file.
  - When the user wants to revert to the file, the delta is reversed to get to the requested version.



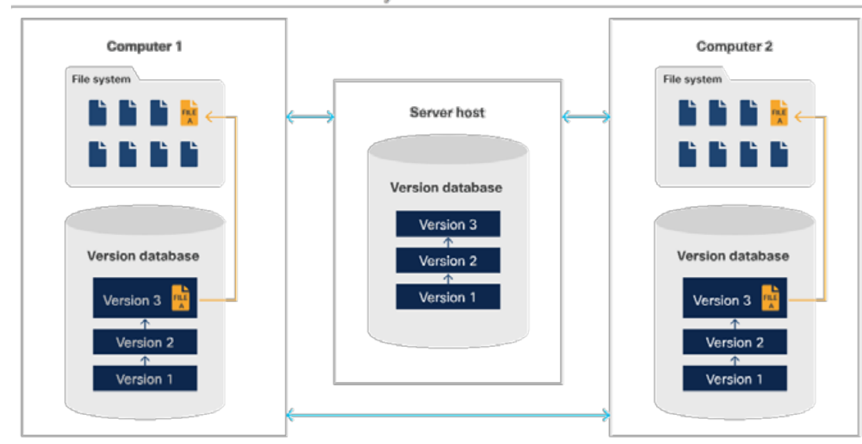
Types of Version Control Systems

- **Centralized Version Control System (CVCS)**
  - CVCS uses a server-client model.
  - The repository is stored in a centralized location, on a server.
  - In CVCS, only one individual can work on a particular file at a time.
  - An individual must check out the file to lock it and make the required changes and check in once done.



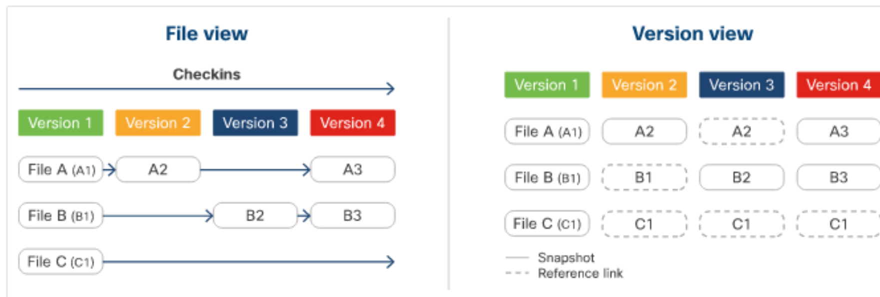
Types of Version Control Systems

- **Distributed Version Control System (DVCS)**
  - DVCS is a peer-to-peer model.
  - The repository can be stored on a client system, but it is usually stored in a repository hosting service.
  - In DVCS, every individual can work on any file, at the same time, because the local file in the working copy is being modified. Hence, locking is not required.
  - When the individual has made the changes, they push the file to the main repository that is in the repository hosting service, and the version control system detects any conflicts between file changes.



Git

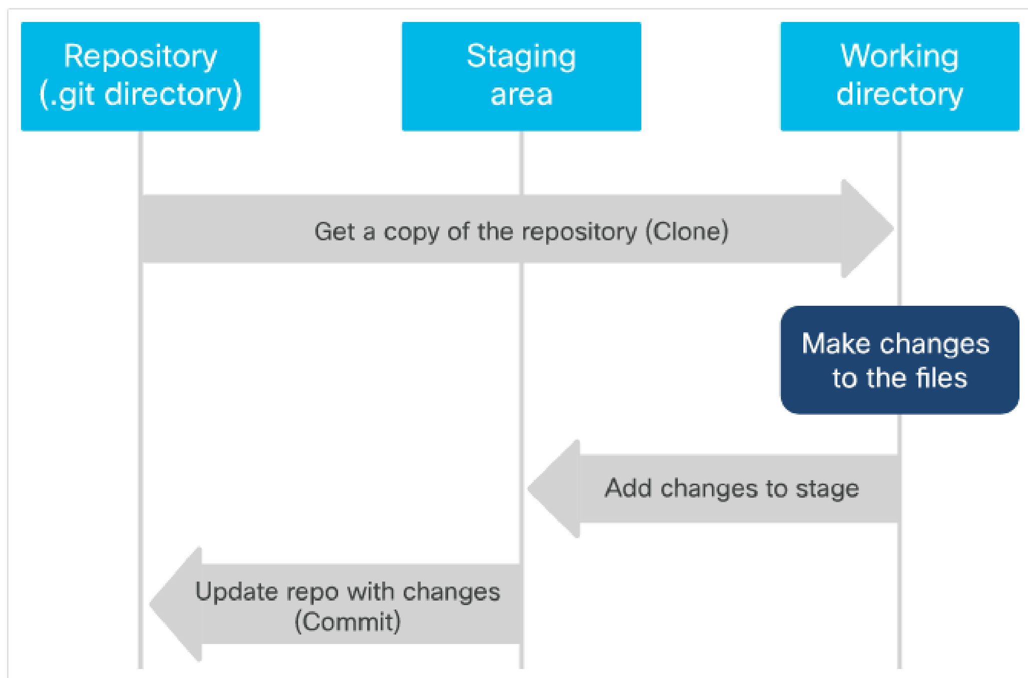
- Git is an open source implementation of a distributed version control system that is currently the latest trend in software development.
- A Git client must be installed on a client machine. It is available for MacOS, Windows, and Linux/Unix.
- One key difference between Git and other version control systems is that Git stores data as snapshots instead of differences (the delta between the current file and the previous version).



- If the file does not change, Git uses a reference link to the last snapshot in the system instead of taking a new and identical snapshot.

Git

- Git is organized by 3s- three stages and three states.
- The three stages are:
  - Repository (the .git directory)
  - Working directory
  - Staging area
- The three states are:
  - Committed
  - Modified
  - Staged

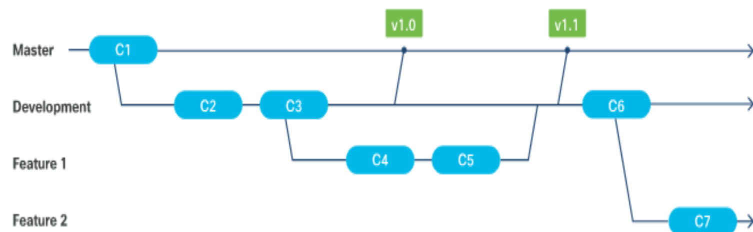


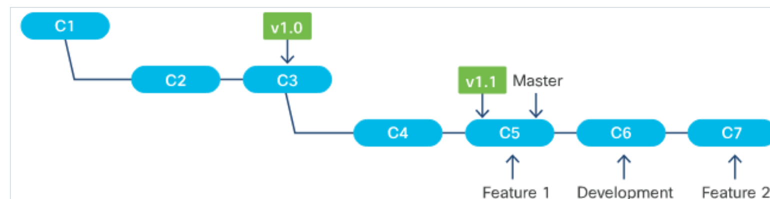
Local vs. Remote Repositories

- Git has two types of repositories:
  - A **local repository** is stored on the file system of a client machine, which is the same one on which the git commands are being executed.
  - A **remote repository** is stored somewhere other than the client machine, usually a server or repository hosting service.
  - A remote repository with Git continues to be a DVCS because the remote repository will contain the full repository, which includes the code and the file history.
- When a client machine clones the repository, it gets the full repository without requiring to lock it, as in a CVCS.
- After the local repository is cloned from the remote repository or the remote repository is created from the local repository, the two repositories are independent of each other until the content changes are applied to the other branch through a manual Git command execution.

What is Branching?

- Branching enables users to work on code independently without affecting the main code in the repository. When a repository is created, the code is automatically put on a branch called Master.
- Branches can be local or remote, and they can be deleted and have their own history, staging area, and working directory.
- Git's branch creation is lightweight, and switching between branches is almost instantaneous.
- When a user goes from one branch to another, the code in their working directory and the files in the staging area change accordingly, but the repository (.git) directories remain unchanged.





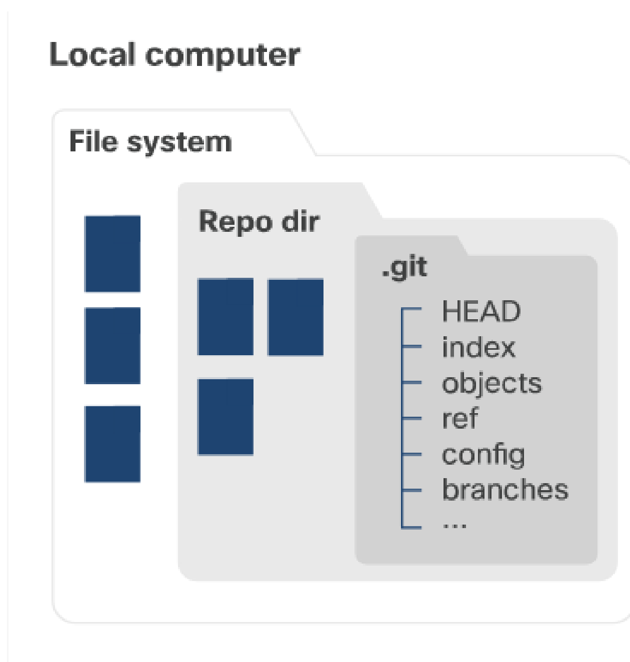
GitHub and Other Providers

- Git and GitHub are not the same.
- While Git is an implementation of distributed version control and provides a command line interface, GitHub is a service provided by Microsoft that implements a repository hosting service with Git.
- In addition to providing the distributed version control and source code management functionality of Git, GitHub provides additional features such as:
  - code review
  - documentation
  - project management
  - bug tracking
  - feature requests
- GitHub introduced the concept of the ‘pull request’, which is a way of formalizing a request by a contributor to review changes such as new code, edits to existing code, etc., in the contributor's branch for inclusion in the project's main or other curated branches.

Git Commands

- **Setting up Git**
- To configure Git, use the --global option to set the initial global settings.

  `git config` --global key value
- Create a New Git Repository
  - Git provides a **`git init`** command to create an empty Git repository, or make an existing folder a Git repository.
  - When a new or existing project becomes a Git repository, a hidden .git directory is created in that project folder.
  - The .git directory is the repository that holds the metadata such as the compressed files, the commit history, and the staging area. In addition, Git also creates the master branch.



Git Commands

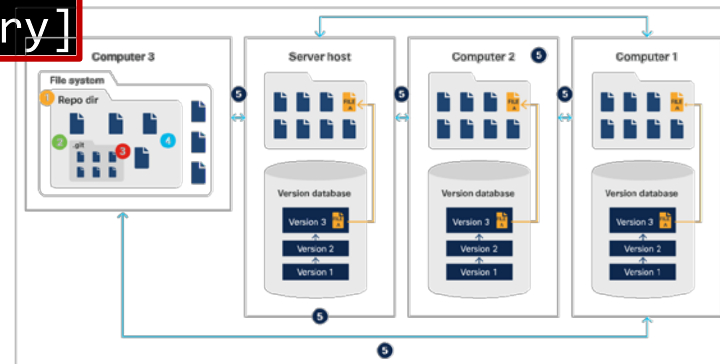
- **Start a new repository**
  - Creates an empty Git repository or makes an existing folder a Git repository.

```
$ git init <project directory>
```

  - where the **<project directory>** is the absolute or relative path to the new or existing project.
  - For a new Git repository, the directory in the provided path will be created first, followed by the creation of the .git directory.
- **Get an Existing Git Repository**

```
$ git clone <repository> [target directory]
```

  - where **<repository>** is the location of the repository to clone.
- Git supports four major transport protocols for accessing the **<repository>:** Local, Secure Shell (SSH), Git, and HTTP.



Git Commands

- **View the Modified Files in the Working Directory**
  - Git provides a git status command to get a list of files that have differences between the working directory and the parent branch.

```
$ git status
```

- **Compare Changes Between Files**
  - Git provides a git diff command that is essentially a generic file comparison tool.

```
$ git diff
```

  - When using the git diff command, the file does not need to be a Git tracked file.

Adding and Removing Files

- **Adding Files to the Staging Area**
  - This command can be used more than once before the Git repository is updated (using commit).
  - Only the files specified in the git command can be added to the staging area
  - To add a single file to the staging area:

```
$ git add <file path>
```

  - To add multiple files to the staging area, where the <file path> is the absolute or relative path of the file to be added to the staging area.

```
$ git add <file path 1> ... <file path n>
```

  - To add all the changed files to the staging area:

```
$ git add
```

Adding and Removing Files

- **Removing Files from the Git Repository**
- There are two ways to remove files from the Git repository:
  - **Option 1**: `git rm` command is used to remove files from the Git repository and add to the staging area.

```
$ git rm
```

- To remove the specified file(s) from the working directory and add the change to the staging area, use the following command:

```
$ git rm <file path 1> ... <file path n>
```

- where `<file path>` is the absolute or relative path of the file to be deleted from the Git repository.

Adding and Removing Files

- To add the specified file(s) to be removed from the staging area without removing the file(s) itself from the working directory, use the following command:

```
$ git rm --cached <file path 1> ... <file path n>
```

  - The `git rm` command will not work if the file is already in the staging area with changes.

- **Option 2**: This option is a two-step process. First use the regular filesystem command to remove the file(s) and then add the file to the stage using the Git command.

```
$ rm <file path 1> ... <file path n>
$ git add <file path 1> ... <file path n>
```

  - This two-step process is equivalent to using the `git rm <file path 1> ... <file path n>` command. Using this option does not allow the file to be preserved in the working directory.

Updating Repositories

- **Updating the Local Repository with the Changes in the Staging Area**
  - This command combines all the content changes in the staging area into a single commit and updates the local Git repository.
  - To commit the changes from the staging area, use the following command:

```
$ git commit
```

  - To commit the changes from the staging area with a message, use the following command:

```
$ git commit -m "<message>"
```

Updating Repositories

Updating the Remote Repository

- Updates the remote Git repository with the content changes from the local Git repository.

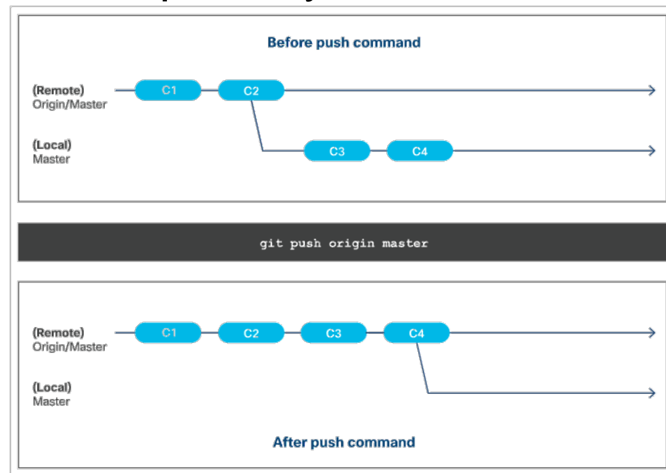
```
$ git push
```

- This command will not execute successfully if there is a conflict with adding the changes from the local Git repository to the remote Git repository.
- To update the contents from the local repository to a particular branch in the remote repository, use the following command:

```
$ git push origin <branch name>
```

- To update the contents from the local repository to the master branch of the remote repository, use the command:

```
$ git push origin master
```



Updating Repositories

- **Updating Your Local Copy of the Repository**
  - Local copies of the Git repository do not automatically get updated when another contributor makes an update to the remote Git repository.
  - Updating the local copy of the repository is a manual step.

```
$ git pull
```

  - When executing the command, the following steps occur:
    - The local repository (.git directory) is updated with the latest commit, file history, and so on from the remote Git repository.
    - The working directory and branch is updated with the latest content from step 1.
    - A single commit is created on the local branch with the changes from step 1.
    - The working directory is updated with the latest content.

Updating Repositories

- **To update the local copy of the Git repository from the parent branch, use the following command:**

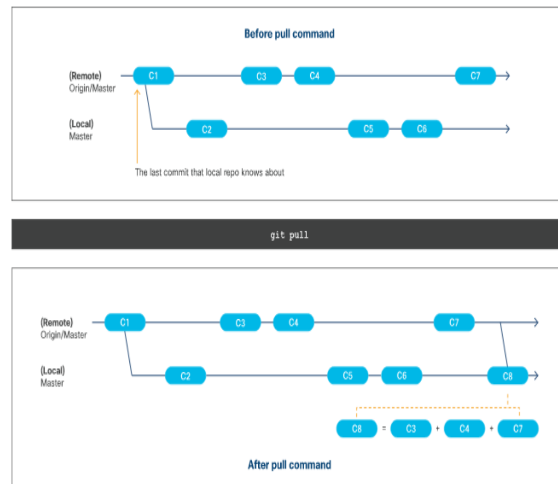
```
$ git pull
```

Or

```
$ git pull origin
```

- To update the local copy of the Git repository from a specific branch, use the following command:

```
$ git pull origin <branch>
```



Branching Features

- **Creating and Deleting a Branch**
  - Option 1: git branch command to list, create, or delete a branch.

```
$ git branch <parent branch> <branch name>
```

  - Option 2: git checkout command to switch branches by updating the working directory with the contents of the branch.

```
$ git checkout -b <parent branch> <branch name>
```

- **Deleting a Branch**
  - To delete a branch, use the following command:

```
$ git branch -d <branch name>
```

- **Get a List of all Branches**
  - To get a list of all the local branches, use the following command:

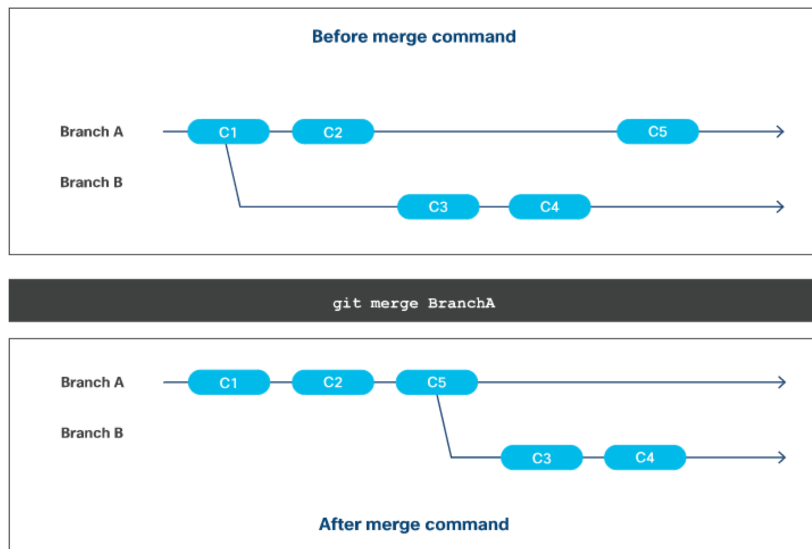
```
$ git branch
```

or

```
$ git branch --list
```

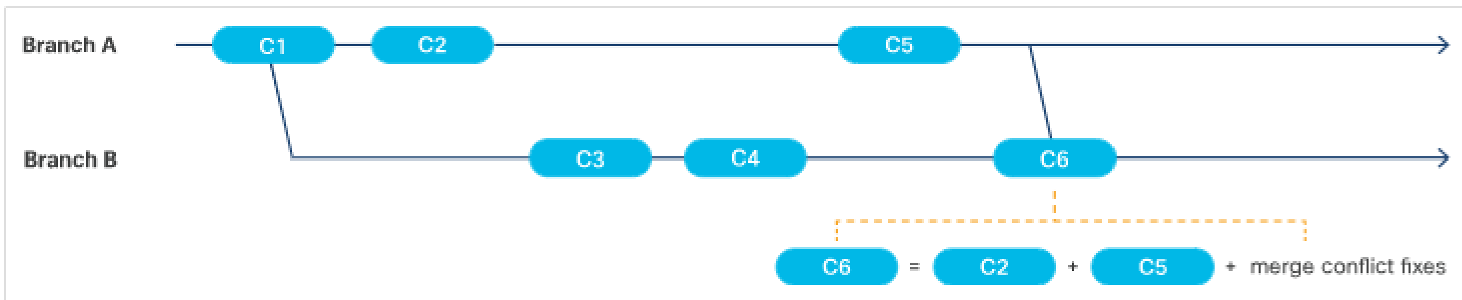
Branching Features

- **Merging Branches**
  - Branches diverge from one another when they are modified after they are created.
  - When Git merges the branch, it takes the changes/commits from the source branch and applies it to the target branch.
  - During a merge, only the target branch is modified.
  - The source branch is untouched and remains the same.



Branching Features

- **Fast-Forward Merge**
  - A fast-forward merge is when the Git algorithm is able to apply the changes/commits from the source branch(es) to the target branch automatically and without any conflicts.
- **Merge Conflicts**
  - A merge conflict is when Git is not able to perform a fast-forward merge because it does not know how to automatically apply the changes from the branches together for the file(s).



Branching Features

- **Performing the Merge**
  - Git provides a git merge command to join two or more branches together.

```
$ git merge
```

To merge a branch into the client's current branch/repository, use the below command:

```
$ git merge <branch name>
```

To merge a branch into a branch that is not the client's current branch/repository, use the following command:

```
$ git checkout <target branch name>
$ git merge <source branch name>
```

  - To merge more than one branch into the client's current branch/repository, use the below command:

```
$ git merge <branch name 1>...<branch name n>
```

.diff Files

- A .diff file is used to show how two different versions of a file have changed.
- By using specific symbols, this file can be read by other systems to interpret how files can be updated.
- The symbols and meanings in a unified diff file are:

Symbol	Meaning
+	Indicates that the line has been added.
-	Indicates that the line has been removed.
/dev/null	Shows that a file has been added or removed.
or "blank"	Gives context lines around changed lines.
@@	A visual indicator that the next block of information is starting. Within the changes for one file, there may be multiple.
index	Displays the commits compared.

3.4 Coding Basics

Methods, Functions, Modules, and Classes

- As the project size and complexity grows, and other developers (and stakeholders) get involved, disciplined methods and best practices are needed to help developers write better code and collaborate around it more easily.

```
# Define the function
def functionName:
  ...blocks of code...
# Call the function
functionName()
```

```
class Circle:
    def __init__(self, radius):
        self.radius = radius
```

Clean Code

- Clean codes are the result of developers trying to make their code easy to read and understand for other developers.
- Clean codes emphasize on standardization, proper organization, modularity, providing inline comments and other characteristics that help make code self-documenting.
- They follow some common principles related to formatting, organization, intuitiveness of components, purpose and reusability.
- Reasons why developers want to write clean code:
  - Easier to understand, more compact, and better-organized.
  - Modular and tends to be easier to test using automated methods such as unit testing frameworks.
  - Standardized and is easier to scan and check using automated tools.
  - It simply looks nicer.

Methods and Functions

- Methods and Functions are blocks of code that perform tasks when executed.
- Functions are standalone code blocks whereas methods are code blocks associated with an object.
- Following are some standard best-practices for determining whether a piece of code should be encapsulated (in a method or function):
  - Code that performs a discrete task, even if it happens only once, may be a candidate for encapsulation.
  - Task code that is used more than once should probably be encapsulated.
- Methods and Functions can be written once and executed as many times as required.
- If used correctly, methods and functions will simplify the code, and reduce the potential for bugs.
- Syntax of a Function in Python:

```
# Define the function
def functionName:
   ...blocks of code...
# Call the function
functionName()
```

Methods and Functions

- **Arguments and Parameters**
  - Arguments and parameters add flexibility to methods and functions.
  - Syntax of a function using arguments and parameters in Python:

```
# Define the function
def functionName(parameter1,...,parameterN):
  # You can use the parameters just like local variables
  ...blocks of code...
# Call the function
functionName("argument1", 4, {"argument3":"3"})
```

Methods and Functions

- **Return Statements**
  - The return statement refers to the return value that is specified using the keyword return followed by a variable or expression. A return statement ends the execution of a function and returns control to the calling function.
  - When a return statement is executed, the value of the return statement is returned and any code below it gets skipped.
  - Syntax of a function with a return statement in Python:

```
# Define the function
def functionName(parameter1,...,parameterN):
  # You can use the parameters just like local variables
  ...blocks of code...
  someVariable = parameter1 * parameter2
  return someVariable
# Call the function
myVariable = functionName("argument1", 4, {"argument3":"3"})
```

Methods and Functions

- **Methods vs. Functions**

Methods	Functions
Methods are code blocks associated with an object, typically for object-oriented programming. The function is packaged in a single Python file.	Functions are standalone code blocks.

Modules

- Developers typically use modules to divide a large project into smaller parts so that the code can be read and understood easily.
- They consists of a set of functions and typically contains an interface for other modules to integrate with.
- A module is packaged as a single file and is expected to work independently.
- Below is a module with a set of functions saved in a script called circleClass.py.

```
# Given a radius value, print the circumference of a circle.
# Formula for a circumference is c = pi * 2 * radius

class Circle:

    def __init__(self, radius):
        self.radius = radius

    def circumference(self):
      pi = 3.14
      circumferenceValue = pi * self.radius * 2
      return circumferenceValue

    def printCircumference(self):
      myCircumference = self.circumference()
      print ("Circumference of a circle with a radius of " + str(self.radius) + " is " +
str(myCircumference))
```

Classes

- In most Object-Orient Programming (OOP) languages, and in Python, classes are a means of bundling data and functionality. Each class declaration defines a new object type.
- Classes may have class variables and object variables.
- New classes may be defined, based on existing, previously defined classes, so that they inherit the properties, data members, and functionality (methods).
- A class may be instantiated (created) multiple times, and each with its own object-specific data attribute values.
- Note: Unlike other OOP languages, in Python, there is no means of creating 'private' class variables or internal methods. However, by convention, methods and variables with a single preceding underscore (_) are considered private and not to be used or referenced outside the class.

```
>>> class Url():
...     def __init__(self, host, prot):
...         self.host = host
...         self.prot = prot
...         self.url = self.prot + "://" + self.host
...
>>> url2 = Url('www.cisco.com', 'http')
```

3.5 Code Review and Testing

What is a Code Review and Why Should You Do This?

- A code review is when developers look over the codebase, a subset of code, or specific code changes and provide feedback. These developers are often called reviewers.
- The code review process only happens after the code changes are complete and tested.
- The goal of code reviews is to make sure that the final code:
  - Is easy to read
  - Is easy to understand
  - Follows coding best practices
  - Uses correct formatting
  - Is free of bugs
  - Has proper comments and documentation
  - Is clean

Types of Code Reviews

- The most common types of code review processes include:
  - **Formal code review**: Developers have a series of meetings to review the whole codebase. It promotes discussion among all the reviewers.
  - **Change-based code review**: Also known as a tool-assisted code review, reviews code that was changed as a result of a bug, user story, feature, commit, and so on.
  - **Over-the-shoulder code review**: A reviewer looks over the shoulder of the developer who wrote the code and provides feedback.
  - **Email pass-around**: It can occur following the automatic emails sent by the source code management systems when a checkin is made.

Testing

- Software testing is classically subdivided into two general categories:
  - **Functional testing** seeks to determine whether the software works correctly. Does it behave as intended in a logical sense, from the lowest levels of detail examined with Unit Testing, to higher levels of complexity explored in Integration Testing?
  - **Non-functional testing** examines usability, performance, security, resiliency, compliance, localization, and many other issues. This type of testing finds out if the software is fit for its purpose, provides the intended value, and minimizes risk.
- Developers capture design requirements as tests and then write software to pass those tests. This is called **Test-Driven Development (TDD)**.

Unit Testing

- Detailed functional testing of small pieces of code (lines, blocks, functions, classes, and other components in isolation) is called Unit Testing.
- These test frameworks are software that allows you to make assertions about testable conditions and determine if these assertions are valid at a point in execution.
- Examples of test frameworks for Python:

PyTest	unittest
• PyTest is handy. It automatically executes any scripts that start with test_ or end with _test.py and within those scripts, automatically executes any functions beginning with 'test_' or 'tests_'. • We can unit test a piece of code by copying it into a file, importing pytest, adding appropriately-named testing functions, saving the file under a filename that also begins with 'tests_,' and running it with PyTest.	• The unittest framework demands a different syntax than PyTest. • For unittest, you need to subclass the built-in TestCase class and test by overriding its built-in methods or adding new methods whose names begin with 'test_'.

Integration Testing

- Integration testing ensures that all the individual units fit together properly to make a complete application.
- Running the code with PyTest produces an output as shown in the image:

```
============================ test session starts ==============================
platform linux2 -- Python 2.7.15+, pytest-3.3.2, py-1.5.2, pluggy-0.6.0
rootdir: /home/ubuntu/deploysample, inifile:
collected 1 item
test_sample_app.py F                                                     [100%]
================================== FAILURES ==================================
_______________________________ test_setconfig ________________________________
    def test_setconfig():
        setUp()
        set_config("TESTVAL")
>       assert get_config() == "ESTVAL"
E       AssertionError: assert 'TESTVAL' == 'ESTVAL'
E         - TESTVAL
E         ? -
E         + ESTVAL
test_sample_app.py:21: AssertionError
------------------------------ Captured log call ------------------------------
connectionpool.py          225 DEBUG    Starting new HTTP connection (1): localhost:80
connectionpool.py          437 DEBUG    http://localhost:80 "GET /get_config HTTP/1.1" 200 7
connectionpool.py          225 DEBUG    Starting new HTTP connection (1): localhost:80
connectionpool.py          437 DEBUG    http://localhost:80 "GET /config_action?dbhost=TESTVAL HTTP/1.1"
200 30
connectionpool.py          225 DEBUG    Starting new HTTP connection (1): localhost:80
connectionpool.py          437 DEBUG    http://localhost:80 "GET /get_config HTTP/1.1" 200 7
=========================== 1 failed in 0.09 seconds ===========================
```

- Note: You can run this script on your VM using pytest. However, understanding the output and fixing any errors is beyond the scope of this course.

Test-Driven Development (TDD)

- If you want to test to validate the application design in light of requirements, implies that you should write the testing code before you write the application code .
- Having expressed the requirements in your testing code, you can then write the application code until it passes the tests that you have created in the testing code.
- The basic pattern of TDD is a five-step, repeating process:
  - Create a new test.
  - Run tests to see if any fail for unexpected reasons.
  - Write application code to pass the new test.
  - Run tests to see if any fail.
  - Refactor and improve application code.

3.6 Understanding Data Formats

Data Formats

- Rest APIs let you exchange information with remote services and equipment.
- The three most popular standard formats for exchanging information with remote APIs are XML, JSON, and YAML.
- Parsing XML, JSON, or YAML is a frequent requirement of interacting with APIs. An oft-encountered pattern in REST API implementations is as follows:
  - Authenticate, usually by POSTing a user/password combination and retrieving an expiring token for use in authenticating subsequent requests.
  - Execute a GET request to a given endpoint (authenticating as required) to retrieve the state of a resource, requesting XML, JSON, or YAML as the output format.
  - Modify the returned XML, JSON, or YAML.
  - Execute a POST (or PUT) to the same endpoint (again, authenticating as required) to change the state of the resource, again requesting XML, JSON, or YAML as the output format and interpreting it as needed to determine if the operation was successful.

XML

- **Extensible Markup Language (XML)** is a generic methodology for wrapping textual data in symmetrical tags to indicate semantics.
- It is a derivative of Structured, Generalized Markup Language (SGML), and also the parent of HyperText Markup Language (HTML). XML filenames typically end in ".xml".
- An Example XML Document

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Instance list -->
<vms>
  <vm>
    <vmid>0101af9811012</vmid>
    <type>t1.nano</type>
  </vm>
  <vm>
    <vmid>0102bg8908023</vmid>
    <type>t1.micro</type>
  </vm>
</vms>
```

XML

- **XML Document Body**: Except the first two lines of a XML document, the remainder of the document is considered as the body.
- **User-Defined Tag Names**: XML tag names are user-defined. If you are composing XML for your own application, pick tag names that clearly express the meaning of data elements, their relationships, and hierarchy.
- **Special Character Encoding**: Data is conveyed in XML as readable text.
- **XML Prologue**: The XML prologue is the first line in an XML file.
- **Comments in XML**: XML files can include comments, using the same commenting convention used in HTML documents.
- **XML Attributes**: XML lets you embed attributes within tags to convey additional information.

XML

- **XML Namespaces:**
  - Namespaces are defined by the IETF and other internet authorities, organizations, and other entities, and their schemas are typically hosted as public documents on the web.
  - Namespaces are identified by Uniform Resource Names (URIs) to make persistent documents reachable without the seeker needing to be concerned about their location.
  - The code example below shows the use of a namespace, defined as the value of an xmlns attribute, to assert that the content of an XML remote procedure call should be interpreted according to the legacy NETCONF 1.0 standard.

```
<rpc message-id="101" xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <kill-session>
    <session-id>4</session-id>
  </kill-session>
</rpc>
```

XML

- **Interpreting XML**
  - In the XML Namespaces example, the structure is represented as a list or one-dimensional array (called 'instances') of objects (each identified as an 'instance' by bracketing tags). Each instance object contains two key-value pairs denoting a unique instance ID and VM server type.
  - A semantically-equivalent Python data structure might be declared as shown below:

```
vms [
  {
    {"vmid": "0101af9811012"},
    {"type": "t1.nano"}
  },
  {
    {"vmid": "0102bg8908023"},
    {"type": "t1.micro"}
  }
]
```

JSON

- JSON, or JavaScript Object Notation, is a data format derived from the way complex object literals are written in JavaScript.
- JSON filenames typically end in ".json."
- Below is a sample JSON file, containing two values that are text strings, one is a boolean value, and two are arrays:

```
{
  "edit-config":
  {
    "default-operation": "merge",
    "test-operation": "set",
    "some-integers": [2,3,5,7,9],
    "a-boolean": true,
    "more-numbers": [2.25E+2,-1.0735],
  }
}
```

JSON

- **JSON Basic Data Types**: JSON basic data types include numbers, strings, Booleans, or nulls.
- **JSON Objects**: As in JavaScript, individual objects in JSON comprise of key/value pairs, which may be surrounded by curly braces { }, individually.
- **JSON Maps and Lists**: In this case, each individual key/value pair does not need its own set of brackets, but the entire object does. JSON compound objects can be deeply-nested, with complex structure. It can also express JavaScript ordered arrays (or 'lists') of data or objects.
- **No Comments in JSON**: Unlike XML and YAML, JSON does not support any kind of standard method for including unparsed comments in code.
- **Whitespace Insignificant**: Whitespace in JSON is not significant, and files can be indented using tabs or spaces as preferred, or not at all.

YAML

- YAML Ain't Markup Language (YAML) is a superset of JSON designed for even easier human readability.
- As a superset of JSON, YAML parsers can generally parse JSON documents (but not vice-versa).
- Hence, YAML is better than JSON at certain tasks, including the ability to embed JSON directly (including quotes) in YAML files.

```yaml
---
edit-config:
  a-boolean: true
  default-operation: merge
  more-numbers:
  - 225.0
  - -1.0735
  some-integers:
  - 2
  - 3
  - 5
  - 7
  - 9
  test-operation: set
...
```

YAML

- **YAML File Structure**: YAML files conventionally open with three dashes (--- alone on a line) and end with three dots (... likewise).
- **YAML Data Types**: YAML basic data types include numbers, strings, Booleans, or nulls.
- **Basic Objects**: In YAML, basic data types are equated to keys.
- **YAML Indentation and File Structure**: YAML indicates its hierarchy using indentation.
- **Maps and Lists**: YAML easily represents more complex data types, such as maps containing multiple key/value pairs and ordered lists.
  - Maps are generally expressed over multiple lines, beginning with a label key and a colon (:), followed by members, indented on subsequent lines:

```
mymap:
  myfirstkey: 5
  mysecondkey: The quick brown fox
```

YAML

- **Lists** (arrays) are represented with optionally-indented members preceded by a single dash and space:

```
mylist:
  - 1
  - 2
  - 3
```

- **Maps** and **lists** can also be represented in a so-called "flow syntax," which looks very much like JavaScript or Python:

```
mymap: { myfirstkey: 5, mysecondkey: The quick brown fox}
mylist: [1, 2, 3]
```

YAML

- **Long Strings**: They are represented using a 'folding' syntax, where linebreaks are presumed to be replaced by spaces when the file is parsed/consumed, or in a non-folding syntax.

```
mylongstring: >
  This is my long string
  which will end up with no linebreaks in it
myotherlongstring: |
  This is my other long string
  which will end up with linebreaks as in the original
```

YAML

- **Comments**: Comments in YAML can be inserted anywhere except in a long string literal, and are preceded by the hash sign and a space.

```
# this is a comment
```

- **More YAML Features**: YAML has many more features, most often encountered when using it in the context of specific languages, like Python, or when converting to JSON or other formats. For example, YAML 1.2 supports schemas and tags, which can be used to disambiguate interpretation of values.
- For example, to force a number to be interpreted as a string, you could use the !!str string, which is part of the YAML "Failsafe" schema:

```
mynumericstring: !!str 0.1415
```

Parsing and Serializing

- **Parsing** means analyzing a message, breaking it into its component parts, and understanding their purposes in context.
- **Serializing** is roughly the opposite of parsing.
- Popular programming languages such as Python generally incorporate easy-to-use parsing functions that can accept data returned by an I/O function and produce a semantically-equivalent internal data structure containing valid typed data.
- On the outbound side, they contain serializers that turn internal data structures into semantically-equivalent messages formatted as character strings.

3.7 Software Development and Design Summary

What Did I Learn in this Module?

- Six phases of SDLC: Requirements & Analysis, Design, Implementation, Testing, Deployment and Maintenance.
- Three popular software development models are Waterfall, Agile, and Lean.
- The MVC design pattern simplifies development of applications that depend on graphic user interfaces.
- Version control maintains history of changes to a file. Types of version control systems: Local, Centralized, and Distributed.

What Did I Learn in this Module?

- Git is an open source implementation of a distributed version control system and has two types of repositories: local and remote.
- Clean code is the result of developers trying to make their code easy to read and understand for other developers.
- Code review involves reviewing a codebase, a subset of code, or specific code change to provide feedback.
- Three most popular standard formats for exchanging information with remote APIs: XML, JSON and YAML.
- Parsing requires analyzing a message, breaking it into its component parts, and understanding their purposes in context. Serializing is roughly the opposite.

New Terms and Commands

- Software Development Life Cycle (SDLC)
- User experience (UX)
- Software Requirement Specification (SRS)
- Agile Scrum
- Lean
- Extreme Programming (XP)
- Feature-Driven Development (FDD)
- Sprints
- Backlog
- User stories
- Scrum Teams
- Model-View-Controller (MVC)
- Centralized Version Control Systems (CVCS)
- Distributed Version Control System (DVCS)
- Git
- Branching
- GitHub
- Arguments
- Parameters
- Object-Orient Programming (OOP)
- Formal Code Review
- Change-Based Code Review
- Over-the-Shoulder Code Review
- Test-Driven Development (TDD)
- Unit Testing
- Software Development Kits (SDKs)
- XML
- JSON
- YAML
- Application Programming Interfaces (APIs)
- REpresentational State Transfer (REST)
- Long Strings
- Parsing
- Serializing

CISCO

Elk Technologies
https://elktech.org
SCOTT STEPHENSON