1. Agile Methodology
    a. Agile Values
        i. Individuals and interactions over processes and tools.
        ii. Working software over comprehensive documentation.
        iii. Customer collaboration over contract negotiation.
        iv. Responding to Change over following a plan.
    b. Key Principles
        i. Customer satisfaction through early and continuous delivery of valuable software.
        ii. Deliver working software frequently, with a preference of shorter timeframe.
        iii. Welcome changing requirement even in development stage.
        iv. Business people and developers must work together daily throughout the project.
    c. Agile in lament terms
        i. Make a list.
        ii. Size things up (gather resources).
        iii. Set priorities.
        iv. Start executing.
        v. Update the plans you go.
    d. Agile Key features
        i. Iterative Development.
        ii. Adaptive Planning.
        iii. Blurred Roles (Single person can have many talents).
        iv. Requirements Change.
        v. Working Software.
2. Fundamental Approaches: Agile
    a. User Stories
        i. User stories describe features.
        ii. The stories are from user point of view.
        iii. These features can be delivered in short units of work.
        iv. They are often documented to facilitate communication.
    b. Estimation
        i. Estimation is difficult but essential.
        ii. Estimation of development time should be practiced well.
    c. Iterations
        i. Stories
        ii. Analysis and Design
        iii. Development
        iv. Test
        v. Working software
        vi. Repeat all steps again
    d. Planning
        i. Combination of user stories and estimation to build a feasible plan for delivery.
3. Flavours of Agile
    a. Scrum

        i. Easy to understand and start
        ii. Very popular
        iii. Not much Engineering
        iv. Can be used in non-IT projects.
    b. Extreme Programming
        i. Detailed Engineering Practices
        ii. IT focused
        iii. Popular with developers
        iv. Upfront testing
        v. Automation
        vi. Evolutionary design
        vii. Continuous integration
4. Full Stack Development
    a. Full Stack
        i. Developing Every part of web application
        ii. Technologies used from mobile and front end (HTML, CSS, JavaScript) and backend logic, security and database models used at the backend.
        iii. Creating a Client Server architecture
        iv. Popular Stack Development (LAMP, LEMP)
    b. Development Environment
        i. As servers don't require a graphical front end, web development is usually don from command line.
        ii. Usage of Git to develop and push code to the server.
5. Flask
    a. Micro framework
        i. Runs on any machine and has few dependencies.
        ii. Python3 is required to set up environment.
        iii. Install Flask using *pip* and run the flask app using *flask run*.
        iv. Create a python programme and use a browser(localhost) to see the app.
    b. App structure
        i. Single module structure
        ii. Package structure baseline
        iii. Package structure for big project
    c. Server-Side Rendering
        i. Use python functions to build html pages
        ii. HTML reference codes to store in a templates or views directory, a rendering functions to build model dynamically.
        iii. Flask uses jinja to do this. Pug, handlebars and typescript are alternatives.
    d. Jinja
        i. Separate Logic and presentation.
        ii. When a request is received flask will look for matching template and convert the template to pure html using named variables in the function.
        iii. Two {{curly braces}} are used to distinguish html from python variables, and jinja does the substitution.

6. MVC architecture
    a. Architectural pattern
        i. Design patterns describe reusable design concepts, particularly in software. They describe how objects are organized to call each other.
        ii. Examples are client-server architecture, pipe and filter, and blackboard architectures.
        iii. Some specific patterns that apply to web applications are Model View Controller, Boundary Control Entity, 3-Tier Architecture and Model View View-Model.
    b. Model-View-Control
        i. The model view controller patter is one of the most popular for server-side web applications.
        ii. The model refers to an object referencing an entity in a database.
        iii. The view is how that object is presented to the user.
        iv. The controller is a linking class that builds the model from the database, prepares the view based on the model, and the updates and saves the models back to the database.
    c. Model View viewmodel
        i. Model View View-Model is a variation of model view controller that is tailor for client-side applications and single page applications. Rather than having a controller compose the view a binder links the view to a viewmodel
        ii. The view presents the current state of the viewmodel
        iii. The viewmodel exposes the data and available operations of the model, and updates the model as required.
        iv. Two-way data-binding links the view and viewmodel without need to link back to the server.
    d. Boundary Control Entity
        i. Boundary Control Entity pattern is often used for enterprise systems and doesn't have strong coupling between data and presentation.
        ii. The boundary object(s) control the interface to the subsystem, and filter requests and responses to objects external to the subsystem.
        iii. The control object processes the requests, update the entity objects and prepare the responses.
        iv. The entity objects represent the data in the system, and link to persistent data sources, like databases.
7. Designing a MVC structure.
    a. Mock Websites
        i. Wireframe drawing show the basic layout and functionality of a user interface.
        ii. There are various tools for building these, or you can draw them by hand.
        iii. A series of wire frame mocks can show the sequence of interfaces used in an application.
        iv. You can also mock the typical http requests and responses your app will serve.
    b. Implementing Models

i. A model is an object that is paired with an entity in a database.
ii. There is an Object Relational Mapping (ORM) linking the data in the database to the models in the application.
iii. The models are only built as needed and update the database as required. Most frameworks include ORM support
iv. To build the models, we first need to set up the database
v. There are relational databases, document databases, graph databases and others
vi. We will focus on relational databases and particularly SQLite, but we will discuss alternatives.

c. Relational Databases.
   i. Relational databases store data as a set of relations, where each relation is represented as a table
   ii. Each row of the table is an entity, and each column of the table is an attribute of that entity
   iii. Every relation has an attribute that is unique for every entity in that relation, called the primary key
   iv. Some relations attributes that are primary keys in other relations. These are called foreign keys.

d. NoSQL (MongoDB)
   i. MongoDB (from humongous) is a free and open-source cross-platform document-oriented database.
   ii. Classified as a NoSQL database, MongoDB avoids the traditional table-based relational database structure in favour of JSON-like documents with dynamic schemas
   iii. As of July 2015, MongoDB is the fourth most popular type of database management system, and the most popular for document stores.
   iv. Schema-less structure
   v. Documents are objects.
   vi. No joins at all.
   vii. Data should be tree like.

e. Document Databases
   i. Document databases don't have tables or schemas. Instead, they consist of Collections of Documents
   ii. Each document in a collection may have different fields. The fields of a document can be another document (a sub-document), but two documents cannot share a subdocument. i.e it is a tree. In Mongo, each document is represented as a JSON object.
   iii. Database - Database is a physical container for collections. Each database gets its own set of files on the file system.
   iv. Collection - Collection is a group of MongoDB documents. It is the equivalent of an RDBMS table. A collection exists within a single database. Collections do not enforce a schema.
   v. Document - A document is a set of key-value pairs. Documents have dynamic schema. Dynamic schema means that documents in the same collection do not need to have the same set of fields or structure.

f. Linking models to app
    i. Now we have a database setup, we would like to link it into our application. We will use SQL-Alchemy for ORM with SQLite. Alternatively, we could use *pymongo* with Mongo or py2neo with Neo4J.
    ii. We need to install flask-sql-alchemy and flask-migrate
    iii. We will keep the database in a file called *app.db*, in the root of our app, and include this in config.py
    iv. Next we update __init__.py to create an SQL-Alchemy object called db, create a migrate object, and import a module called models (which we will write)
    v. The models classes define the database schema
g. SQL-Alchemy Models
    i. To build a model we import db (the instance of SQL-Alchemy) and our models are then all defined to be subclasses of *db.Model*
    ii. To see what these modules are doing, you can find the source code in the virtual environment directory.
    iii. *db.Column* is a class used to specify the type and constraints of each column in the table.
    iv. *db.relationship* is a function that defines attributes based on a database relationship.
h. Database initialisation
    i. This allows us to define the database schema, but we still need to link it to the database. Flask provides some utilities to do this.
    ii. *flask db init* will initialise a database to synchronize with the models you have defined.
    iii. *flask db migrate* will use alembic to create a migration script that applies changes to the database.
    iv. *flask db upgrade* applies that script to the database (and downgrade to roll the changes back.)
    v. This allows us to keep the database schema and the models in sync.
i. Linking in with views and controllers
    i. We can now respond to requests for data, by building models from the database, and then populating views with the data.
    ii. As the code is getting complex, it is a good idea to have a Controllers.py class, rather than handling everything in routes.py.