1. Here are some good reasons for using Docker:
    - Similar environments
    - Application as a whole package
      Using the Docker images makes possible to package all of your application and dependencies, making the distribution easy because it won't be necessary to send an extent documentation explaining how to configure the required infrastructure to allow the execution, just make the image available in a repository and grant the access to the user, so the user can download the build that will be executed with no problems.
    - Standardization and replication
    - Common language between infrastructure and development
      The syntax used to parameterize the Docker images and environments can be considered as a common language between areas that usually don't dialogue well.
    - Community
2. it is possible do make available the complete application package in a repository, and this "final product" be easily used without needing a complete configuration. Just one configuration file and one command can be enough to start an application build as a Docker image.
3. To create the required isolation in the process, Docker uses the kernel functionality, called namespaces, that creates isolated environments between containers: the processes of a running application will not have access to the resources of another one. Unless it is expressly enabled in the configuration of each environment.
4. To avoid the exhaustion of machine resources due to one isolated environment, Docker uses the cgroups feature from kernel. This makes possible the coexistence os different containers in the same host, without one affecting the other for overusing shared resources
5. Docker Engine: It's the base software of the solution. It is both the daemon responsible for the containers and the client used to send commands to daemon.
6. Docker Compose: It's the tool responsible for defining and executing multiple containers based on definition files.
7. Docker Machine: It's the tool that enables to create and keep Docker environments in virtual machines, cloud environments and even in a physical machine.
8. docker container stop meu_python
   In the command above, in case there was a container named meu_python running, it would receive a SIGTERM signal and, if it was not turned off, it would receive a SIGKILL after 10 seconds
9. Making a parallel with the concept of object orientation, image is the class and container is the object. The image is the infrastructure abstraction with reading only status, from where the container is going to be instantiated.
10. AUFS is a unification file system. It is responsible for managing multiple directories, stacking them up, and provide a single and unified view, as if they all together were only one directory.
    This single directory is used to present the container and works as if it was a single common file system. Each directory used in the stack corresponds to one layer. And that's how Docker unifies them and provides reutilization amongst containers.

Because the same directory that corresponds to the image can be set up in several stacks of several containers.

11. What Docker calls network, in fact, is an abstraction created to ease the data communication management between containers and untie the external knots of the Docker environment.
12. Standard networks on Docker:
    - Bridge
      Each container created on Docker is associated to an specific network. This is the standard network to any container, unless we associate, explicitly, another network to it. The network gives to the container an interface that makes a bridge with the docker0 interface of the Docker host. This interface receives, automatically, the next address available in the IP network 172.17.0.0/16.
      All containers in this network can communicate via TCP/IP protocol. If you know which is the IP address of the container you wish to connect, it is possible to send data to it. After all, they are all in the same IP network (172.17.0.0/16). A detail worth noticing: as the IPs are assigned automatically, it'n not a trivial task to discover which is the IP of the destination container. To help on this location, Docker provides, at the moment of creating a container, the option "-link".
      It's important to emphasize that "-link" is an outdated option and its use is discouraged. We'll explain this feature only for understanding the legacy. This function was replace with a built-in DNS on Docker and it doesn't work for Docker standard networks, only for networks created by the user. The option "-link" is responsible for associating the destination container IP to its name.
    - None
      This network aims to isolate the container regarding external communications. The network doesn't get any interface to external communication. The only interface of the IP network will be the localhost.
      This network, usually, is used for containers that only manipulate files, with no need of sending them to another place using the network (Ex.: backup container uses the volumes of the database container to do the dump and will be used in the data retention process.
    - Host
      This network has the objective of delivering into the container all the interfaces existent on Docker host. In a way, it can speed up the package delivery, once there's no bridge on the way of the messages. But usually this overhead is minimum and the use of a bridge can be important to security and management of traffic
    - Networks set by user
      Docker allows the user to create networks. These networks are associated to the element the Docker calls network driver.
      Each network created per user must be linked to a given driver. And in case you didn't create your own driver, you must choose amongst the drivers provided by Docker
13. Network drivers provided by Docker:

- Bridge
  This is the network driver more simple to use, for requires little configuration. The network created by the user using the bridge driver is similar to the Docker standard network named “bridge”.
  One more point that deserves attention: Docker has a standard network called “bridge” that uses a driver also called “bridge”. Maybe, because of this, the confusion only gets bigger. But it is important to make clear that they are distinct.
  The networks created by the user with the bridge driver have all features described in the standard network, called bridge. However, it has additional features. Amongst one of the features: the network created by the user doesn’t need to user the old “-link” option. Because every network created by the user with the bridge driver will be able to user the Docker internal DNS that automatically associates every container names of this network to its respective IPs from the corresponding IP network.
  To make it clearer: the containers using the standard bridge network will not be able to enjoy the Docker internal DNS feature. In case you are using this network, it is necessary to specify the “-link” legacy for translating the names in IP addresses dynamically allocated on Docker.

14. two services: the one that runs in daemon mode, in background, called Docker Host, responsible for viabilization of containers on the kernel Linux; and the client, that we’ll call Docker client, responsible for getting commands from the user and translating them into management of Docker Host.
    Each Docker client is configured to connect itself to a given Docker host and, at this moment, Docker machine takes the action, for it enables the automatization of access configuration choice of Docker client to distinct Docker hosts.
    The Docker machine enables the use of several different environments just changing the client configuration to the desired Docker host: basically, modify some environment variables
15. Aiming to define a series of best practices common to modern web applications, some developers from Heroku wrote the 12Factor app (https://12factor.net/) manifesto, counting on a wide experience in developing web applications.
    “The Twelve-Factor app” (12factor) is a manifesto with a series of best practices for building software using automation declarative formats, maximizing the portability and minimizing divergencies amongst execution environments, allowing the deployment in modern cloud platforms and facilitating scalability. Thus, applications are build stateless and connected to any infrastructure services combination to data retention (database, queue, cache memory and similar).
    Here are examples - https://github.com/gomex/exemplo-12factor-docker
16. 1 best practice - Codebase
    each application must have only one code base and, from it, must be deployed in different environments. It’s important to emphasize that this practice is also part of the Continuous Integration (CI) practices. Traditionally, most part of continuous integration systems have, as a starting point, a code base that is built and, later, deployed to development, test and production.

17. 2 best practice - Dependencies
    Explicitly declare and isolate dependencies
18. 3 best practice - Config
    The goal of the best practice is to make feasible the application configuration without the need of changing the code. Since the application behavior varies according to the environment where is executed, the configurations must consider the environment.
19. 4 best practice - Backing services
    Treat backing services as attached resources.
    A backing service is any service the app consumes over the network as part of its normal operation. Examples include datastores (such as MySQL or CouchDB), messaging/queueing systems (such as RabbitMQ or Beanstalkd), SMTP services for outbound email (such as Postfix), and caching systems (such as Memcached).
    The code for a twelve-factor app makes no distinction between local and third party services. To the app, both are attached resources, accessed via a URL or other locator/credentials stored in the config. A deploy of the twelve-factor app should be able to swap out a local MySQL database with one managed by a third party (such as Amazon RDS) without any changes to the app's code. Likewise, a local SMTP server could be swapped with a third-party SMTP service (such as Postmark) without code changes. In both cases, only the resource handle in the config needs to change.
20. 5 best practice - Build, release, run
    12factor indicates that the base code, to be put into production, needs to go through three phases:
    - Build - convert the repository code into executable package. In this process we obtain the dependencies, compile the code's binaries and actives.
    - Release - the package produced in the build phase is combined with the configuration. The result is the whole environment, configured and ready to run.
    - Run (also known as "runtime") - begins running the release (application + configuration of that environment), based on the specific configurations of the required environment.
21. 6 best practice - Processes
    The best practice says that 12factor application processes are stateless (don't store state) and share-nothing. Any data that need to persist must be stored in stateful support service, usually used in a database.
22. 7 best practice - Port binding
    Export services via port binding.
    even that Docker allows using more than one port per container, the best practice emphasizes that you should only use one binding port per application.
23. 8 best practice - Concurrency
    it's important that the service is ready to escalate. The solution must be able to initiate new processes of the same application if necessary, without affecting the product. Aiming to avoid any kind of problem in service scalability, the best practice says that the applications must support concurrent executions, and when a process is in execution, instantiating another one in parallel and attend the service, without any loss.

24. 9 best practice - Disposability
    applications should be able to quickly remove defective processes. The applications must be disposable; in other words, shutting down one of its instances must not affect the solution as a whole. Another important detail of the best practice is to enable the code to shut down “graciously” and restart with no errors. Thus, when hearing a SIGTERM the code must finish any requirement in progress and then shut down the process with no problems and quickly, allowing that another process is quickly attended as well
    In order to finish up first performs a SIGTERM and wait 10 seconds so the application shuts down by itself; otherwise, it sends a SIGKILL that shuts down the process abruptly. This time out is configurable. In case you wish to change it, just use the parameter “-t” or “–timeout“. Check an example:
    docker-compose stop -t 5
25. 10 best practice - Development/production parity
    Keep development, staging, and production as similar as possible
26. 11 best practice - Logs
    The best practice says that the applications should not manage or route log files, but should be deposited without any buffer to the standard output (STDOUT). Thus, an infrastructure external to the application - platform - must manage, collect and format the logs output to future reading. This is really important when the application is running in several instances. With Docker, such task becomes easy for Docker already collects standard output logs and send them to some of the several log drivers. The driver can be configured in the container initialization in order to group the logs in the log remote service, such as syslog.
27. 12 best practice - Admin processes
    The best practice recommends admin processes executed in environments similar to the ones used in the running code, following all of the practices presented so far.
28. A fast way of cleaning containers and images is by using the following command:
    docker system prune
    With this command you will remove:
    - All the containers not in use at the moment
    - All the volumes not in use by at least one container
    - All the networks not in use by at least one container
    - Every dangling images

29. Use a “linter”
    “Linter” is a tool that provides tips and warning on some source code.
30. It’s usual to put other files, such as documentation, in the same directory of ‘Dockerfile’; to improve the building performance, delete files and directories creating a *dockerignore* file in the same directory. This file works similarly to ‘.gitignore’. Using it helps to minimize the building context *docker build*.
    Avoid adding packages and unnecessary extra dependencies to the application and minimize complexity, image size, building time and attack surface.
    Also minimize the layer amount: whenever possible, group up various commands. However, take in consideration the volatility and maintenance of these layers.

31. In most cases, run only one process per container. Decoupling applications in several container eases up horizontal scalability, reuse and monitoring of containers.
32. Choose COPY over ADD
    ADD - It allows to download url files and automatically extract files of known formats (tar, gzip, bzip2, etc.).
    On the other hand 'COPY' is a simpler command to put files and folders of the building path inside the Docker image. Thus, choose 'COPY' unless you are absolutely sure that 'ADD' is necessary
33. Use a image with smallest base
34. Use the layer building cache
    ```
    COPY package.json /app/
    RUN npm install
    COPY . /app
    ```
35. Clean on the same layer
    ```
    RUN apt-get update && \
    apt-get install -y curl python-pip && \
    pip install requests && \
    apt-get remove -y python-pip curl && \
    rm -rf /var/lib/apt/lists/*
    ```
    Remember that the cleaning must be made in the same layer (command 'RUN'). Otherwise, data will be persisted on this layer and removing them later will not have the same effect in the final image size.
36. Be careful while adding data to a volume on Dockerfile
    Avoid adding a lot of data in a folder and then turn it into a 'VOLUME' only when starting the container, because you can slow down the loading.
37. never expose public ports. However, expose the application' standard ports privately.
    ```
    # public and private mapping, avoid
    EXPOSE 80:8080
    # only private
    EXPOSE 80
    ```
38. Container. This virtualization model is on the operational system level; that is, different from a virtual machine, a container does not view the whole machine, it's just a running process in a kernel shared amongst all other containers. It uses the namespace to provide the due isolation of RAM memory, processing, disk and network access. Even when shared in the same kernel, this running process views the use of a dedicated operational system.
39. Useful commands
    Remove all inactive containers
    docker container prune
    Stop all containers
    docker stop $(docker ps -q)
    Remove all local images
    docker image prune
    Remove "orphan" volumes
    docker volume prune
    Shows use of containers resources running

docker stats $(docker ps --format {{.Names}})

40.

41.