Tema

Desplegar aplicaciones Python

Manual del Estudiante

Maria del Carmen Sierra Fernández

Contenido

Empaquetado de aplicaciones Python 3

Pensando en el despliegue 3
Empaquetado de librerías y herramientas de Python 3
Módulos de Python 4
Distribuciones fuente de Python 4
Distribuciones binarias de Python 4
Otras opciones de empaquetado de aplicaciones de Python 5
Dependientes de un framework 5
Plataformas de servicio 5
Navegadores web y aplicaciones móviles 6
Depende de un entorno con Python preinstalado 6
Depende de un entorno de distribución de software separado 6
Crear un ejecutable de Python 6
Traer su propio espacio de usuario 7
Traer su propio núcleo 7

Entornos Virtuales y Paquetes 8

Virtualenv 8
Creando Entornos Virtuales 9
Crear un entorno Python aislado 9
Manejando paquetes con pip 10

Crear un ejecutable con pyInstaller 11

Instalación 11
¿Cómo funciona pyinstaller? 12
Opciones de compilación 13

Docker 16

Crear una aplicación Python y empaquetarla en un contenedor Docker 17
Ejemplos de ficheros Dockerfile 18
Ejemplo de una aplicación Python en un contenedor Docker en AWS 19

Empaquetado de aplicaciones Python

Como lenguaje de programación de propósito general, Python está diseñado para usarse de muchas maneras. Puede crear sitios web, robots industriales, un juego, y mucho más, todo utilizando la misma tecnología central.

La flexibilidad de Python es la razón por la cual el primer paso en cada proyecto de Python debe ser pensar en la audiencia del proyecto y el entorno correspondiente donde se ejecutará el proyecto. Puede parecer extraño pensar en empaquetar antes de escribir código, pero este proceso nos ayuda a evitar futuros dolores de cabeza.

Pensando en el despliegue

Los paquetes existen para ser instalados (o implementados), por lo que antes de empaquetar cualquier cosa, debemos tener algunas respuestas a las siguientes preguntas de implementación:

¿Quiénes son los usuarios del software? ¿El software será instalado por otros desarrolladores que realizan desarrollo de software, personas de operaciones en un centro de datos o un grupo menos experto en software?

¿El software está diseñado para ejecutarse en servidores, ordenadores de escritorio, clientes móviles (teléfonos, tabletas, etc.) o integrado en dispositivos dedicados?

¿EL software se instala individualmente o en grandes lotes de implementación?

El empaquetado tiene que ver con el entorno de destino y la experiencia de implementación. Hay muchas respuestas a las preguntas anteriores y cada combinación de circunstancias tiene sus propias soluciones. En este tema conoceremos algunas tecnologías de empaque que nos ayudarán a implementar nuestro proyectos.

Empaquetado de librerías y herramientas de Python

Herramientas como PyPI, setup.py, Wheel entre otras; son solo algunas de las herramientas que proporciona el ecosistema de Python para el empaquetado y distribución de software python.

- **PyPi (Python Package Index):** es un repositorio de software para el lenguaje de programación Python. Básicamente es un repositorio, como podría ser github, pero especializado para Python.
- **setup.py** es un script contenido dentro de los paquetes o librerías para asegurar su completa y correcta instalación.
- **Wheel**: es un estándar de distribución de Python

Módulos de Python

Un archivo de Python, siempre que solo se base en la biblioteca estándar, se puede redistribuir y reutilizar. También deberá asegurarse de que esté escrito para la versión correcta de Python y solo se base en la biblioteca estándar.

Esto es excelente para compartir secuencias de comandos y fragmentos simples entre personas que tienen versiones de Python compatibles (por ejemplo, por correo electrónico, StackOverflow o GitHub gists).

Sin embargo, este patrón no se escalará para proyectos que constan de varios archivos, necesitan librerías adicionales o necesitan una versión específica de Python, de ahí las opciones a continuación.

Distribuciones fuente de Python

Si su código consta de varios archivos de Python, generalmente está organizado en una estructura de directorio. Cualquier directorio que contenga archivos de Python puede incluir sentencias de importación de módulos y/o paquetes.

Dado que los paquetes constan de varios archivos, son más difíciles de distribuir. La mayoría de los protocolos admiten la transferencia de un solo archivo a la vez.

Siempre que el código no contenga más que código Python puro y estemos seguros de que el entorno de implementación es compatible con su versión de Python, podemos usar las herramientas de empaquetado nativas de Python para crear un paquete de distribución `sdist`.

Los `sdists` de Python son archivos comprimidos (archivos .tar.gz) que contienen uno o más paquetes o módulos. Si su código es Python puro y solo depende de otros paquetes de Python.

Si el código Python utiliza librería de terceros como en el caso de numpy, debemos utilizar otra forma de distribución.

Distribuciones binarias de Python

Gran parte del poder práctico de Python proviene de su capacidad para integrarse con el ecosistema de software, en particular, librerías escritas en C, C++, Fortran, Rust y otros lenguajes.

No todos los desarrolladores tienen las herramientas o la experiencia adecuadas para crear estos componentes escritos en estos lenguajes compilados, por lo que Python creó Wheel, un formato de paquete diseñado para enviar librerías con artefactos compilados. De hecho, el instalador de paquetes de Python, `pip`, siempre prefiere `wheels` porque la instalación

siempre es más rápida, por lo que incluso los paquetes de Python puro funcionan mejor con `wheels`.

Las distribuciones binarias son mejores cuando vienen con distribuciones de origen para que coincidan con cada sistema operativo. Predeterminado para publicar archivos `sdist` y `wheel` juntos, a menos que esté creando artefactos para un caso de uso muy específico en el que sabe que el destinatario solo necesita uno u otro.

- **.py** para módulos locales
- **sdist** para paquetes de Python puro
- **Wheel** para paquetes python

Python y PyPI facilitan la carga conjunta de `wheels` y `sdists`. Simplemente siga el tutorial Empaquetado de proyectos de Python.

Otras opciones de empaquetado de aplicaciones de Python

Hasta ahora solo hemos discutido las herramientas de distribución nativas de Python. Estos enfoques integrados solo tienen como objetivo entornos que tienen Python y una audiencia que sabe cómo instalar paquetes de Python. Con la variedad de sistemas operativos, configuraciones y personas estas herramientas se dirige a una audiencia de desarrolladores.

El paquete nativo de Python está diseñado principalmente para distribuir código reutilizable, llamado librerías, entre desarrolladores.

Las librerías son bloques de construcción, no aplicaciones completas. Para la distribución de aplicaciones, existe todo un nuevo mundo de tecnologías. A continuación, conoceremos otras opciones de empaquetado de aplicaciones según sus dependencias en el entorno de destino.

Dependientes de un framework

Algunos tipos de aplicaciones de Python, como backends de sitios web y otros servicios de red, son lo suficientemente comunes como para tener frameworks que permitan su desarrollo y empaquetado. Otros tipos de aplicaciones, como las interfaces web dinámicas y los clientes móviles, son lo suficientemente complejas donde esto frameworks nos facilitan el trabajo.

Plataformas de servicio

Si está desarrollando para una "Plataforma como servicio" o "PaaS" como Heroku o Google App Engine, existen las siguiente guías de empaquetado.

- Heroku
- Google App Engine
- PythonAnywhere
- OpenShift

- Frameworks “sin servidor” como Zappa

En todas estas configuraciones, la plataforma se encarga del empaquetado y la implementación. La mayoría del software no se ajusta a una de estas plantillas, por ello existen otras opciones.

Navegadores web y aplicaciones móviles

Podemos escribir una aplicación móvil o una interfaz de aplicación web en Python. Si bien el lenguaje puede resultar familiar, las prácticas de empaquetado e implementación son completamente nuevas. Para ello, existen frameworks y guías de empaquetado como:

- Kivy (Kivy: Cross-platform Python Framework for NUI Development)
- Beeware (Write once. Deploy everywhere.— BeeWare)
- Brython
- Flexx (interfaces gráficas)

Depende de un entorno con Python preinstalado

Si tiene Python instalado en su ordenador, podemos utilizar:

- PEX (ejecutable de Python)
- zipapp
- shiv

Depende de un entorno de distribución de software separado

Durante mucho tiempo, muchos sistemas operativos, incluidos Mac y Windows, carecían de una gestión de paquetes integrada. Solo recientemente estos sistemas operativos obtuvieron las llamadas "tiendas de aplicaciones", pero incluso aquellas se centran en aplicaciones de consumo y ofrecen poco a los desarrolladores.

Los desarrolladores buscaron soluciones durante mucho tiempo y, en esta lucha, surgieron con sus propias soluciones de administración de paquetes, como `Homebrew`. La alternativa más relevante para los desarrolladores de Python es un ecosistema de paquetes llamado Anaconda. `Anaconda` se basa en Python y es cada vez más común en entornos académicos, analíticos y otros entornos orientados a datos, incluso llegando a entornos orientados a servidores.

Crear un ejecutable de Python

Cada sistema operativo admite de forma nativa uno o más formatos de programa que pueden ejecutar de forma nativa.

Hay muchas técnicas y tecnologías que convierten un programa de Python en uno de estos formatos, la mayoría de los cuales implican incrustar el intérprete de Python y cualquier otra dependencia en un único archivo ejecutable.

Este enfoque, llamado `freezing`, ofrece una amplia compatibilidad y una experiencia de usuario perfecta, aunque a menudo requiere múltiples tecnologías y una buena cantidad de esfuerzo.

Algunas herramientas, que nos permiten crear un ejecutable son:

- pyInstaller: Multiplataforma
- cx_Freeze: Multiplataforma
- constructor: para instaladores de línea de comandos
- py2exe: solo para sistema operativo Windows
- py2app: solo Mac
- Osnap - Windows y Mac
- Pynsist: solo para Windows

La mayoría implican implementaciones de un solo usuario. Para aplicaciones de servidor de varios componentes, podemos utilizar `Chef Omnibus`.

Traer su propio espacio de usuario

Se puede configurar una cantidad cada vez mayor de sistemas operativos, incluidos Linux, Mac OS y Windows, para ejecutar aplicaciones empaquetadas como imágenes livianas, utilizando un arreglo relativamente moderno que a menudo se denomina virtualización a nivel del sistema operativo o crear contenedores.

Estas técnicas son en su mayoría independientes de Python, porque empaquetan sistemas de archivos completos del sistema operativo, no solo Python o paquetes de Python.

La adopción es más amplia entre los servidores Linux, donde se originó la tecnología y donde las siguientes tecnologías funcionan mejor:

- AppImage
- Docker
- Flatpak
- Snapcraft

Traer su propio núcleo

La mayoría de los sistemas operativos admiten alguna forma de virtualización clásica, ejecutando aplicaciones empaquetadas como imágenes que contienen un sistema operativo propio completo. Ejecutar estas máquinas virtuales, o VM, es un enfoque maduro, generalizado en entornos de centros de datos.

Estas técnicas se reservan principalmente para implementaciones a mayor escala en centros de datos, aunque ciertas aplicaciones complejas pueden beneficiarse de este paquete. Las tecnologías son independientes de Python e incluyen:

- Vagrant
- VHD, AMI y otros formatos
- OpenStack: un sistema de administración de la nube en Python, con amplia compatibilidad con VM

Traer su propio hardware

La forma más integral de enviar su software sería enviarlo ya instalado en algún hardware. De esta manera, el usuario de su software solo necesitaría electricidad.

Mientras que las máquinas virtuales están reservadas principalmente para los expertos en tecnología, puede encontrar dispositivos de hardware que utilizan todos, desde los centros de datos más avanzados hasta usuarios sin experiencia.

Incruste su código en un hardware Adafruit, MicroPython o más potente que ejecute Python, luego envíelo al centro de datos o a los hogares de sus usuario.

Entornos Virtuales y Paquetes

Las aplicaciones en Python la mayoría de las veces hacen uso de paquetes y módulos que no forman parte de la librería estándar. Las aplicaciones a veces necesitan una versión específica de una librería, debido a que dicha aplicación requiere que un bug particular haya sido solucionado o bien la aplicación ha sido escrita usando una versión obsoleta de la interfaz de la librería.

Esto significa que tal vez no sea posible para una instalación de Python cumplir los requerimientos de todas las aplicaciones. Si la aplicación A necesita la versión 1.0 de un módulo particular y la aplicación B necesita la versión 2.0, entonces los requerimientos entran en conflicto e instalar la versión 1.0 o 2.0 dejará una de las aplicaciones sin funcionar.

La solución a este problema es crear un entorno virtual, un directorio que contiene una instalación de Python de una versión en particular, además de unos cuantos paquetes adicionales.

Virtualenv

`virtualenv` es una herramienta que se utiliza para crear entornos Python aislados. Crea una carpeta que contiene todos los ejecutables necesarios para usar los paquetes que necesitaría un proyecto de Python.

Podemos instalarlo con pip:

```
pip install virtualenv
pip install virtualenv --upgrade
```

Actualizar la versión de pip

```
python -m pip install --upgrade pip
```

Para verificar la instalación, podemos utilizar el siguiente comando:

```
virtualenv --version
```

Creando Entornos Virtuales

Para crear un entorno virtual, debemos primero decidir en que carpeta queremos crearlo y ejecuta el módulo `venv` como script con la ruta a la carpeta:

```
python3 -m venv myenv
```

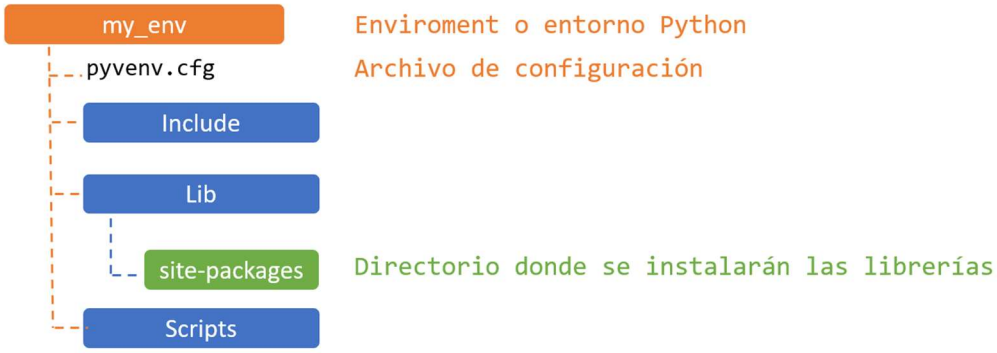
Esto creará el directorio `myenv` si no existe y los subdirectorios que contendrán una copia del intérprete de Python y varios archivos de soporte.

Una vez creado el entorno virtual, podemos activarlo desde la terminal. Nos ubicamos en el directorio `Scripts` dentro de la carpeta creada para el entorno virtual y ejecutamos el fichero `actívate.bat`

```
…\myenv\Scripts\activate.bat
```

Crear un entorno Python aislado

1. Ejecutamos la aplicación Símbolo del Sistema de Windows. Escribimos `cmd` desde la opción buscar de Windows.
   - Verificamos que tenemos instalado el `virtualenv`. Para instalarlo, podemos ejecutar desde el `cmd` el comando: `pip install virtualenv`
2. Nos ubicamos en el directorio donde queremos crear el **entorno virtual del proyecto**: En nuestro caso: `cd C:\Proyectos_XYZ`
3. Para crear nuestro entorno virtual `my_env`, ejecutamos el comando: `python -m venv my_env`
4. Verificamos que la estructura de carpetas de nuestro entorno virtual se ha creado:



5. Utilizando el comando `dir`, podemos verificar la creación de la estructura:
   `C:\Proyectos_XYZ> dir`
6. Activar el entorno virtual
   - Nos ubicamos en el directorio donde se encuentra el fichero `activate.bat` y ejecutamos el siguiente comando:
     `C:\Proyectos_XYZ\my_env\Scripts>activate.bat`
   - Para validar que el entorno esta activo, debemos observar que el prompt ha cambiado a: `(my_env) C:\ Proyectos_XYZ\>`

Manejando paquetes con pip

Una vez creado nuestro entorno virtual y activarlo, podemos instalar, actualizar y eliminar paquetes usando `pip`. `pip` tiene varios subcomandos: install, uninstall, freeze, etc. Lo primero que debemos hacer, es actualizar la versión de pip utilizando el comando:

```
python -m pip install --upgrade pip
```

Podemos instalar la última versión de un paquete especificando el nombre del paquete:

```
(myenv)> python -m pip install novas
```

También podemos instalar una versión específica de un paquete ingresando el nombre del paquete seguido de == y el número de versión:

```
(myenv)> python -m pip install requests==2.6.0
```

Si re-ejecutamos el comando, pip detectará que la versión ya está instalada y no hará nada. Se puede ingresar un número de versión diferente para instalarlo, o se puede ejecutar `pip install --upgrade` para actualizar el paquete a la última versión:

```
(myenv)> python -m pip install --upgrade requests
```

Otras opciones:

- `pip uninstall` seguido de uno o varios nombres de paquetes eliminará los paquetes del entorno virtual.
- `pip show` mostrará información de un paquete en particular: `(myenv)> pip show requests`
- `pip list` mostrará todos los paquetes instalados en el entorno virtual: (myenv)> pip list
- `pip freeze` retorna una lista de paquetes instalados similar, pero el formato de salida es el requerido por pip install. Una convención común es poner esta lista en un archivo `requirements.txt`: `(myenv)> pip freeze > requirements.txt`

El archivo `requirements.txt` puede ser agregado a nuestro control de versiones y distribuido como parte de la aplicación. Los usuarios pueden entonces instalar todos los paquetes necesarios con `install -r`:

```
(myenv)> python -m pip install -r requirements.txt
```

Crear un ejecutable con pyInstaller

PyInstaller agrupa una aplicación de Python y todas sus dependencias en un solo paquete. El usuario puede ejecutar la aplicación empaquetada sin necesidad de instalar el intérprete de Python o módulo.

PyInstaller lee nuestro script o programa escrito en Python, analiza su código para descubrir los módulos y librerías necesita para ejecutarse. Luego recopila copias de todos esos archivos, incluido el intérprete de Python y los coloca en una sola carpeta u, opcionalmente, en un solo archivo ejecutable.

PyInstaller es compatible con Python 3.6 o posterior, y agrupa correctamente los principales paquetes de Python, como numpy, PyQt, Django, wxPython y otros.

PyInstaller se puede ejecutar en Windows, Mac OS X y GNU/Linux. Sin embargo, no es un compilador cruzado: para crear una aplicación de Windows, necesitamos ejecutar PyInstaller en Windows; para hacer una aplicación Linux, debemos crear el instalador en Linux, etc.

Instalación

PyInstaller está disponible en PyPI, es decir, podemos instalarlo a través de pip:

```
pip install pyinstaller
```

PyInstaller se puede instalar con una instalación predeterminada de Python, pero lo recomendable es crear un entorno virtual para el proyecto que deseamos empaquetar e instalar.

Para actualizar la versión instalada utilice: `pip install --upgrade pyinstaller`

¿Cómo funciona pyinstaller?

PyInstaller funciona leyendo nuestro programa Python, analiza todas las importaciones que realiza y agrupa copias de las importaciones de nuestro programa. Por ejemplo, si queremos crear un ejecutable de nuestra aplicación `main.py`, ejecutamos:

```
pyinstaller main.py
```

Después de analizar el código y descubrir todas las librerías y módulos que utiliza, PyInstaller genera un archivo `.spec`, este archivo incluye detalles sobre cómo debe empaquetarse la aplicación. La primera vez que ejecutamos PyInstaller se generará un archivo de especificaciones con algunos valores predeterminados.

Finalmente, PyInstaller genera un ejecutable de la aplicación, incluyendo todas sus dependencias. Cuando termina, aparecerá una subcarpeta llamada `dist` (de forma predeterminada, podemos especificar un nombre diferente) en el directorio del proyecto. Este a su vez, contiene el directorio de la aplicación: donde encontraremos un archivo `.exe` para ejecutar, junto con todas las librerías y otros archivos complementarios necesarios.

Todo lo que necesitamos hacer para distribuir nuestro programa o aplicación, es empaquetar este directorio como un archivo `.zip`. Este paquete deberá extraerse en un directorio donde el usuario tenga permisos de escritura para poder ejecutarse.

Veamos un ejemplo:

1. **Instalar la librería pyinstaller:** `pip install pyinstaller`
2. **Creamos un proyecto `ProyectoEjecutable` desde PyCharm**, marcamos crear fichero main.py, vamos a utilizar el código básico que crea pycharm para este ejemplo. A este código le agregaremos un `time.sleep` para que veamos mejor la ejecución de nuestro porgrama:

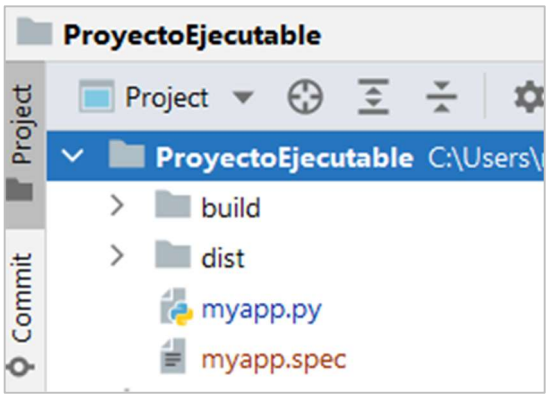
```
# This is a sample Python script.

import time

def print_hi(name):
    print(f'Hi, {name}')

if __name__ == '__main__':
    texto = input("Introduzca un texto: ")
    print_hi(texto)
    time.sleep(10)
```

3. **Crear el ejecutable:** Desde la terminal ejecutamos el comando:

```
pyinstaller --onedir --console main.py
```

```
pyinstaller --onedir --console main.py –name myapp.py
```

observamos que en directorio de trabajo se ha creado la siguiente estructura:



4. Desde el explorador de archivo ejecutamos el fichero `myapp.exe` ubicado en la carpeta `./dist/myapp`

Opciones de compilación

PyInstaller viene con muchas opciones que se pueden proporcionar como archivos de especificaciones u opciones CLI. A continuación, mostraremos algunas de las opciones más comunes y útiles:

`--nombre:` Cambia el nombre de tu ejecutable.

```
pyinstaller main.py --name appcalcular
```

`--onefile:` empaqueta toda la aplicación en un único fichero. Esta opción puede crear fichero muy grandes.

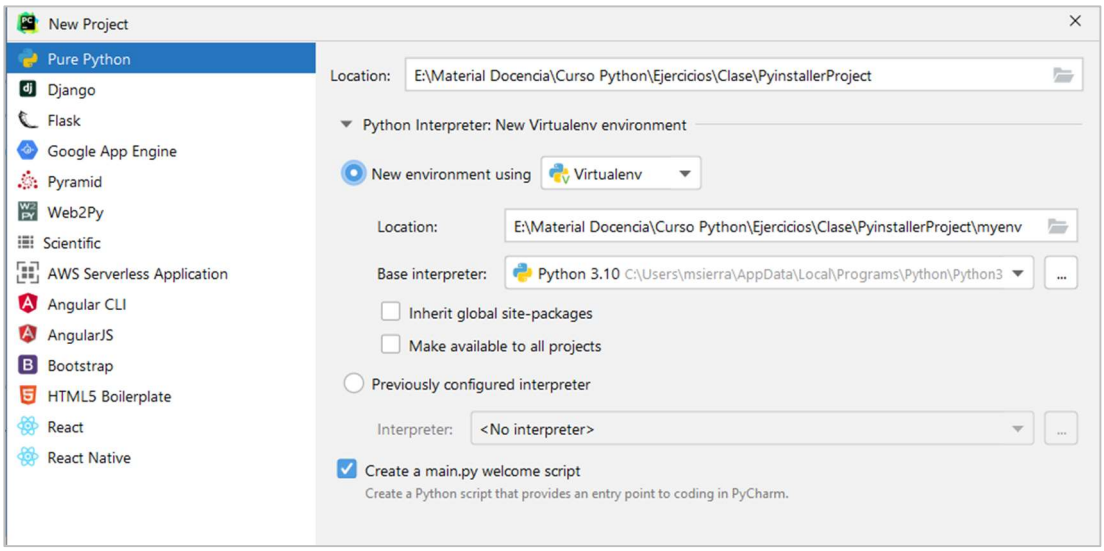
```
pyinstaller --onefile myapp.py
```

`-w:` evita abrir automáticamente una ventana de consola, cuando ejecutamos aplicaciones que tiene una GUI.

```
pyinstaller myapp.py -w
```

Ejemplo:

1. Crear el proyecto `PyinstallerProject` con un entorno virtual



2. Modificar el fichero main.py

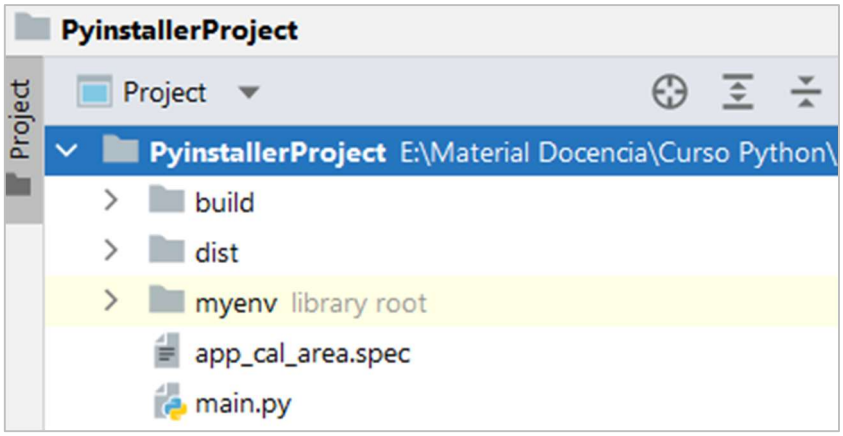
```
from tkinter import *


def limpiar_label():
    label = Label(root, text="
")
    label.place(x=50, y=30)


def saludar():
    limpiar_label()
    label = Label(root, text="Bienvenido")
    label.place(x=50, y=30)


def calcular():
    limpiar_label()
    a = base.get() * altura.get()
    label = Label(root, text=f"El área del triangulo es: {a} ")
    label.place(x=50, y=30)


if __name__ == '__main__':
    # Ventana principal
    root = Tk()
    root.geometry('400x200')
    root.resizable(False, False)
    root.title("Mi programa Python")

    # Variables
    base = IntVar(value=0)
    altura = IntVar(value=0)
```

```
    # Etiquetas y entradas
    etiqueta = Label(root, text="Calcular el área de un
Triangulo!", font=('calibre', 14, 'bold')).pack()

    etiquetabase = Label(root, text="Base").place(x=40, y=60)
    entrada_base = Entry(root, textvariable=base,
width=30).place(x=110, y=60)

    etiquetaaltura = Label(root, text="Altura").place(x=40,
y=80)
    entrada_altura = Entry(root, textvariable=altura,
width=30).place(x=110, y=100)

    # Botones
    botonSaludo = Button(root, text="Saludar")
    botonSaludo.place(x=110, y=150)

    botonCalcular = Button(root, text="Calcular")
    botonCalcular.place(x=170, y=150)

    botonCerrar = Button(root, text="Cerrar",
command=root.quit)
    botonCerrar.place(x=230, y=150)

    botonCalcular.configure(command=calcular)
    botonSaludo.configure(command=saludar)

    root.mainloop()
```

3. Ejecutar desde la terminal el siguiente comando:

```
pyinstaller main.py --name app_cal_area -w
```

4. Verifiquemos la estructura y el ejecutable generado.

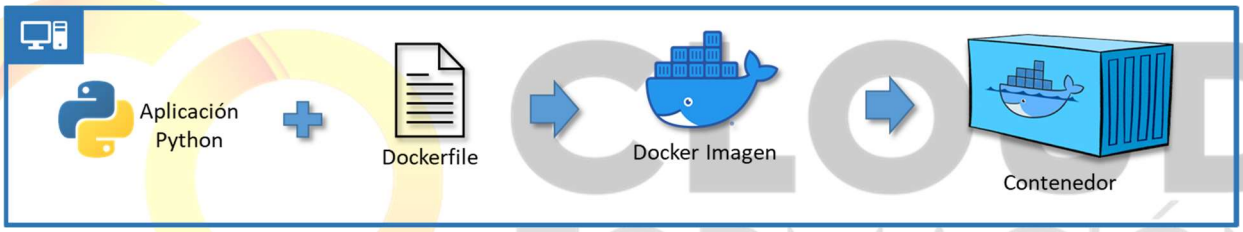


Docker

Docker es un proyecto de código abierto para automatizar la implementación de aplicaciones como contenedores portátiles y autosuficientes que se pueden ejecutar en la nube (Azure, AWS, etc.) o localmente. Es muy potente y versátil.

- Es una plataforma de software que usando contenedores posibilita empaquetar el software y sus dependencias de forma que se pueda usar de la misma forma en diferentes sistemas operativos
- Es un sistema de virtualización basado en contenedores.
- Es una plataforma de software que permite crear, probar e implementar aplicaciones rápidamente.

Docker empaqueta software en unidades estandarizadas llamadas **contenedores** que incluyen todo lo necesario para que el software se ejecute (librerías, herramientas de sistema, código, etc.). Lo cual nos permite implementar aplicaciones rápidamente en cualquier entorno con la certeza que el código se ejecutará.

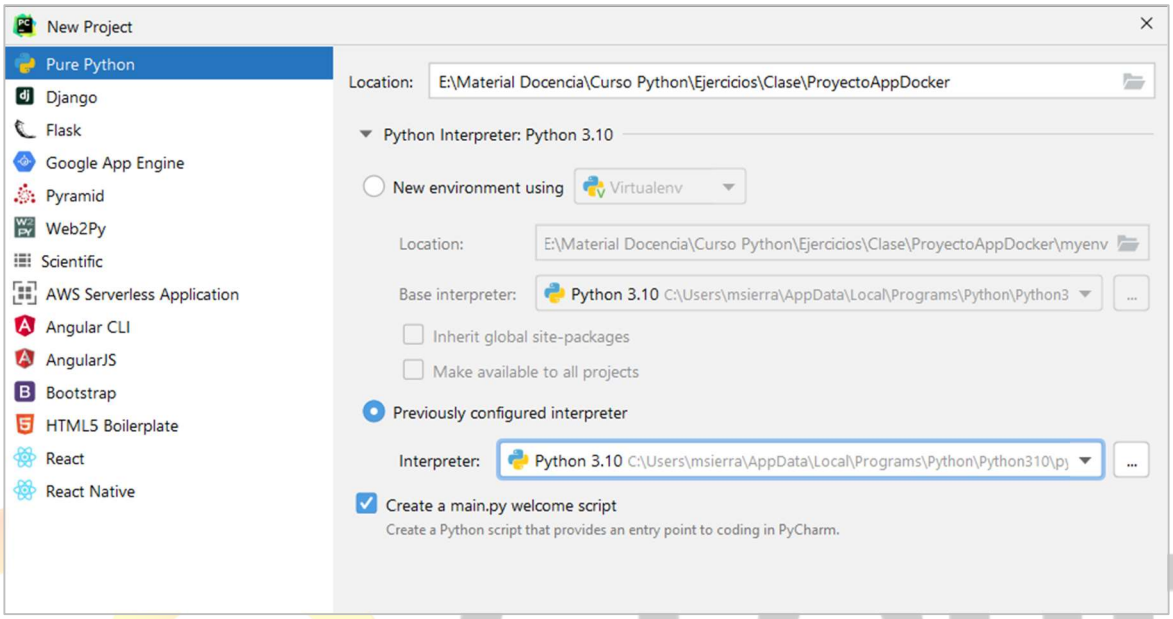


Algunos conceptos:

- **Imagen de contenedor**: Es un paquete con todas las dependencias y la información necesarias para crear un contenedor.
- **Dockerfile**: Es un archivo de texto que contiene instrucciones sobre cómo compilar una imagen de Docker. Es como un script por lotes; la primera línea indica la **imagen base** con la que se comienza y, después, deben seguirse las instrucciones para instalar programas necesarios, copiar archivos, etc., hasta obtener el entorno de trabajo que se necesita.
- **Compilación**: Es la acción de crear una **imagen** de contenedor basada en la información y el contexto que proporciona su Dockerfile, así como archivos adicionales en la carpeta en que se crea la imagen. Puede compilar imágenes con el siguiente comando de Docker: docker build
- **Contenedor**: Es una instancia de una imagen de Docker. Un contenedor representa la ejecución de una sola aplicación, proceso o servicio. Está formado por el contenido de una imagen de Docker, un entorno de ejecución y un conjunto estándar de instrucciones.

Crear una aplicación Python y empaquetarla en un contenedor Docker

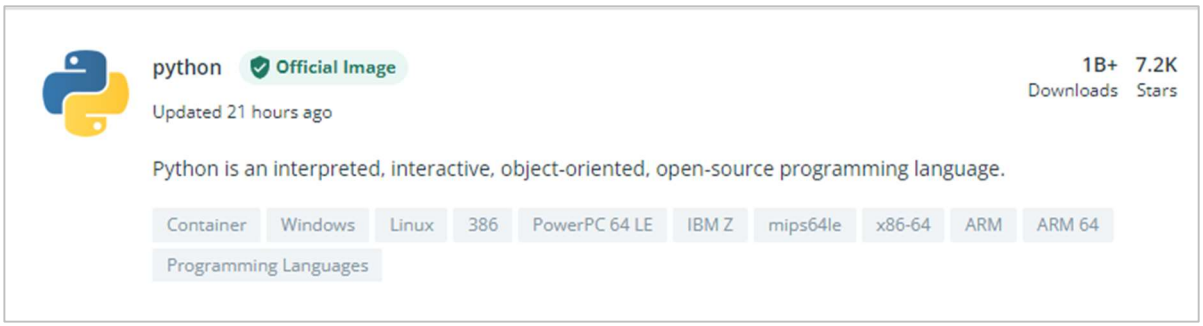
1. **Instalar Docker** en tu ordenador
2. **Crear nuestro proyecto Python** `ProyectoAppDocker` (Este proyecto puede contener aplicación principal Python, módulos, paquetes, enviroment virtual)



3. **Crear el fichero** `Dockerfile` **(New – File):** En el directorio de trabajo creamos un fichero de texto con el nombre Dockerfile.

   Lo primero que hay que hacer cuando creamos un Dockerfile es preguntarnos qué queremos hacer. Nuestro objetivo aquí es ejecutar nuestro código Python. Para hacer esto, nuestro Docker debe contener todas las dependencias necesarias para lanzar Python. Un Linux (Ubuntu) con Python instalado debería ser suficiente.

   Vamos a acceder a la web DockerHub, donde tenemos varias imágenes prediseñadas para ahorrar tiempo (por ejemplo, todas las imágenes de Linux o lenguajes de programación). En nuestro caso, utilizaremos la imagen oficial de Python, creada para ejecutar aplicaciones desarrolladas en Python.



   Nuestro fichero Dockerfile quedaría de la siguiente manera:

```
FROM python:latest
COPY main.py .
CMD [ "python", "./main.py" ]
```

4. **Crear la imagen:** Una vez creado el fichero Dockerfile, creamos la imagen en Docker, para ello debemos ejecutar los siguientes pasos:
    a. Abrir la terminal y nos ubicamos en el directorio de la aplicación `ProyectoAppDocker`.
    b. Crear la imagen `app_docker`, ejecutando el comando:

```
docker build -t app_docker .
```

    c. Lo que hace el comando anterior es buscar el fichero `Dockerfile` en el directorio actual, construye la imagen con el nombre de etiqueta especificado con La opción `-t`.
5. **Ejecutar la imagen creada app_docker**: este paso nos permite ejecutar la imagen localmente y validar que funciona correctamente:

```
docker run -it app_docker
```

Ejemplos de ficheros Dockerfile

Este ejemplo utiliza la imagen Amazon Linux, a la cual le instalamos Python y las librerías boto3 y pymongo.

```
FROM amazonlinux:latest
RUN yum -y install wich unzip python3 pip3
RUN pip3 install boto3
RUN pip3 install pymongo

WORKDIR /app_docker
COPY ./modulo_aws ./modulo_aws
COPY main.py .
CMD ["python3", "main.py"]
```

El siguiente ejemplo utiliza un fichero `requirements.txt`, el cual contiene el listado de todas las librerías que necesitamos instalar para ejecutar nuestra aplicación.

```
FROM python:3

RUN mkdir /my_app
WORKDIR /my_app

COPY requirements.txt /my_app
RUN pip install -r requirements.txt

COPY . /my_app
CMD [ "python", "./main.py" ]
```

El fichero `requirements.txt`, podemos creado ejecutando desde la terminal el comando:

```
pip3 freeze >> requirements.txt
```

Crea el fichero con la lista de librerías utilizadas

```
beautifulsoup4==4.9.3
binarytree==6.3.0
bleach==3.2.1
boto3==1.20.20
botocore==1.23.20
```

Ejemplo de una aplicación Python en un contenedor Docker en AWS

Los servicios de AWS como: AWS Fargate, Amazon ECS, Amazon EKS y AWS Batch nos permiten ejecutar y administrar contenedores de Docker a escala. (AWS Amazon - Docker, 2022)

La siguiente imagen, nos muestra una solución implementada con **Amazon Elastic Container Registry (AWS ECR)** y **Amazon Elastic Container Service (AWS ECS)**, donde AWS ECR nos permite acceder a los repositorios e imágenes de un contenedor, mientras que AWS ECS nos permite gestionar e implementar el contenedor.

Para ejecutar la aplicación del contenedor podemos utilizar **AWS Fargate Task**, crear un servicio o ejecutarla manualmente.