Python:
Conceptos de Aplicaciones en Producción

PYTHON: ENTORNOS AISLADOS

Recordando

Si usamos el intérprete interactivo, todo lo que creamos se pierde el momento en que la sesión se cierra.

Un programa en general vivirá dentro de un archivo o script que podamos modificar o cambiar a nuestro gusto.

A medida que agreguemos complejidad, quizás empecemos a crear funciones que reutilicemos en diferentes sitios de nuestro programa.

Recordando Paquetes

Una forma de agrupar los módulos de manera estructurada.
Un proyecto de Python que define un paquete simplemente requiere un árbol de carpetas. Esta jerarquía se accede en el código mediante puntos.

Por ejemplo para acceder al módulo **eco** uno puede usar

import sonido.efectos.eco

```
sonido/                          Paquete de nivel más alto
      __init__.py                Script de inicialización de este paquete
      formatos/                  Subpaquete
              __init__.py
              mp3.py
              ...
      efectos/                   Subpaquete
              __init__.py
              eco.py
```

Recordando: Dependencias

Python posee en un ecosistema muy grande de paquetes terceros, beneficiando a todos los que necesiten proveer funcionalidades avanzadas o especializadas.

El comando inferior, instalará el paquete deseado desde el repositorio público de paquetes

python -m pip install AlgunPaquete

Adicionalmente, se puede especificar las versiones deseadas en caso de ser necesario.

python -m pip install AlgunPaquete==1.0.4

Problema

A medida que un proyecto crece las dependencias suelen volverse muchas.

Rastrear los paquetes externos dentro del código es algo muy ineficiente en caso de querer preparar una aplicación desde cero.

¿Podemos mantener una lista de dependencias de mejor manera?

requirements

Gestores de paquetes en Python permiten generalmente definir archivos que detallen la lista de dependencias de un proyecto, especificando las mismas características que uno usaría cuando instala las mismas manualmente.

Es común definirlas en un archivo **requirements.txt,** y **requirements-dev.txt** para dependencias que solo se necesiten durante el ciclo de desarrollo de un proyecto.

A pesar de que podemos usar cualquier nombre, esta es una práctica estandarizada.

requirements

Un archivo de ejemplo

requirements.txt

```
pandas
numpy>0.5
matplotlib
algunalibreria==1.0.0
```

requirements-dev.txt

```
pytest
black
```

Como podemos ver, podemos definir las dependencias, inclusive especificando rangos específicos soportados por **pip**. Para instalar, simplemente podemos indicar el archivo:

```
py -m pip install -r requirements.txt
```

requirements

Recomendación: Siempre incluir los requisitos en el código de fuente de su proyecto para permitir la gestión eficiente de las dependencias de sus proyectos.

requirements

A pesar de que esto permite tener mejor reproducibilidad, particularmente al desarrollar un proyecto de manera distribuida. Hay un potencial problema adicional:

Si no se definen las versiones explícitamente, las versiones de las dependencias instaladas en diferentes entornos pueden ser diferentes, o pueden cambiar y dar lugar a comportamientos inesperados.

requirements

Solución: Congelar la versión de las librerías de manera explícita.

Garantiza que siempre tengamos mejor reproducibilidad del entorno, muy útil para capturar regresiones al actualizar versiones de manera controlada.

Opción 1: Congelar requisitos

Mediante **pip freeze**, podemos congelar la versión de **todas** las librerías instaladas en todos los niveles de la aplicación de manera explícita.

```
python -m pip freeze > requirements-frozen.txt
```

Garantiza que siempre tengamos 100% reproducibilidad del entorno, muy útil para capturar regresiones al actualizar versiones independientes.

requirements

requirements.txt

```
pandas
```

requirements-frozen.txt

```
numpy==2.1.1
pandas==2.2.2
python-dateutil==2.9.0.post0
pytz==2024.2
six==1.16.0
tzdata==2024.1
```

Problema: Actualizar es un proceso manual.

Opción 2: Actualización controlada

Mediante **pip-tools**, mantenemos una lista de dependencias **requirements.in** con las versiones ***aceptables,*** *y* generamos dinámicamente un **requirements.txt** de manera controlada

```
python -m pip install pip-tools
python -m piptools compile requirements.in
```

Una ventaja es que, además, actualizará las versiones antes de congelarlas de acuerdo con las reglas de aceptabilidad del archivo de entrada.

```
python -m piptools compile requirements.in --upgrade
```

pip-compile - pip-tools documentation v7.4.2.dev57

requirements

requirements.in

```
pandas>2
```

requirements.txt

```
#
# This file is autogenerated by pip-compile with Python 3.10
# by the following command:
#
#    pip-compile requirements.in
#
numpy==2.1.1
   # via pandas
pandas==2.2.2
   # via -r requirements.in
python-dateutil==2.9.0.post0
   # via pandas
pytz==2024.2
   # via pandas
six==1.16.0
   # via python-dateutil
tzdata==2024.1
   # via pandas
```

Recomendación

¡Siempre mantenerse al tanto de las versiones de nuestras librerías, prevenir la **deuda técnica** a medida que se publican nuevas versiones!

Otro Problema

Como vimos anteriormente, cuando las dependencias se instalan por pip, automáticamente definirá la ubicación como el sitio defecto del sistema.

Es decir, una única librería estará disponible para todas las aplicaciones que la requieran dentro del mismo sistema.

- Limitamos a una versión para todas las aplicaciones
- Perdemos control si alguien cambia la dependencia del sistema
- Poca flexibilidad, si no tenemos acceso a la carpeta centralizada

Solución: Entornos aislados

Una forma de gestionar todas las dependencias y requisitos para una aplicación dentro de un contexto específico, e independiente de las demás.

Con esto podemos tener varias aplicaciones y cada una gestiona sus propias, potencialmente diferentes versiones de manera más consistente.

Solución: Entornos aislados

Muchas posibilidades, una de ellas por ejemplo mediante contenedores (e.g. Docker), sin embargo este es un mundo aparte que requiere conceptos no cubiertos en este curso,

La más común, es la forma predeterminada de Python conocida como entornos virtuales, y que se pueden crear de manera muy sencilla con la herramienta **venv**

Python Virtual Environments: A Primer – Real Python

venv: virtual environment

La herramienta para crear entornos virtuales venv está disponible en Python por defecto en sus últimas versiones.

El siguiente comando permite definir un entorno virtual en la carpeta en la que se invoca.

```
python -m venv venv
```

El comando simplemente creará una carpeta llamada venv en el directorio actual, en la que se instalarán todas las dependencias, incluida una copia individual del intérprete de Python.

Python Virtual Environments: A Primer – Real Python

venv: virtual environment

Una vez creado el entorno, es importante activarlo.

Activarlo simplemente significa cambiar el contexto para que Python se ejecute dentro de esa carpeta.

```
PS> venv\Scripts\Activate.ps1
(venv) PS>
```

En Windows

```
$ source venv/bin/activate
 venv/bin/activate (venv) $
```

En Linux/Sistemas Unix

Python Virtual Environments: A Primer – Real Python

venv: virtual environment

Una vez activado el entorno virtual, la instalación de paquetes, y la ejecución del código será realizada exclusivamente en aquel contexto.

Desactivar el entorno es importante cuando se desee dejar de trabajar en el proyecto actual. Esto se puede realizar mediante el siguiente comando:

deactivate

Python Virtual Environments: A Primer – Real Python

Importancia

Casi siempre leemos que Python recomienda el uso de entornos virtuales para gestión de proyectos.

¿Por qué?

Simplemente porque Python no es el mejor lenguaje en cuanto a gestión de dependencias. Si no somos específicos, pip usará la ubicación por defecto del sistema.

Problemas

- Polución de los paquetes del sistema, incluidos aquellos potencialmente requeridos por utilidades base.
- Conflictos de versiones
- Falta de garantías en cuanto a de reproducción de problemas
- Privilegios del sistema

¿Cómo funciona?

Es muy simple!

Un entorno virtual simplemente crea una estructura de carpetas en las que se incluye una copia del intérprete, y la carpeta donde se instalarán los paquetes externos de Python.

El script de activación le indica a su contexto actual (por ejemplo su sesión del terminal), que Python vive dentro de esta carpeta, ignorando completamente a la instalación del sistema.

Alternativas

- **virtualenv**, el proyecto original que inspiró la creación de venv
- **Pipenv,** un proyecto similar pero que funciona de manera parecida a las dependencias en Node/Javascript
- **Conda** , una solución similar para muchos lenguajes, no solo Python.

Pero **venv** viene por defecto así que es muy útil si no necesitamos comportamientos especiales en nuestra aplicación.

Alternativas

- **contenedores**

En las siguientes clases veremos como esto se puede llevar al siguiente nivel a través de contenedores que permiten virtualizar el entorno completo de ejecución y facilitan el empaquetado y desarrollo de servicios complejos.

Python:
Conceptos de aplicaciones en producción

PYTHON: TESTING

Testing o Verificación

Testing es evaluar nuestro código ante condiciones que verifiquen que su comportamiento sea el esperado dados los requisitos funcionales del mismo.

Un proyecto extenso contendrá necesariamente errores o bugs, pero minimizar su riesgo es algo muy valioso ya que previene que con el pasar del tiempo estos afecten a nuestros usuarios de manera crítica, y adicionalmente reducir costos!

¿Reducir costos?

Contratar un ingeniero o un programador tiene su costo. Si este se emplea en arreglar bugs, es una forma muy poco eficiente de gastar dinero!

Importancia

En industrias clásicas, el control de calidad se vuelve evidente. Por ejemplo, en cuanto a productos alimenticios.

Sin embargo, pensemos en los riesgos potenciales asociados al software!

Consideremos por ejemplo el caso de Nissan, que en su momento tuvo que retirar más de 1 millón de autos debido a fallas en su software de detección de accidentes, o, Bloomberg cuya Terminal usada por inversores tuvo un error debido a un glitch de software, generando pérdidas a muchos clientes del mercado bursátil.

What is Software Testing? Definition, Basics & Types in Software Engineering (guru99.com)

Importancia

Mediante testing podemos detectar errores en la arquitectura del programa, malas decisiones, funcionalidad incorrecta, vulnerabilidades de seguridad, etc.

Buenas prácticas

Existen muchas buenas prácticas en el proceso de testing, las mismas que permiten asegurar un proceso optimizado de verificación del software.

La mejor práctica consiste en la automatización de la mayoría de tareas posibles, en especial en proyectos de amplia escala.

Tipos de test

Funcionales:

Verifican que la función del proyecto sea la definida por sus requisitos. Por ejemplo, comprobar que las funciones evaluadas devuelvan los valores esperados en diferentes escenarios.

Los tests de unidad e integración son ejemplos de tests funcionales.

Tipos de test

No Funcionales:

Verifican que el entorno y capacidades del sistema sean capaces de soportar los requisitos funcionales del mismo. Por ejemplo, asegurarnos que un servicio web sea capaz de soportar una ráfaga de muchos usuarios, o que los usuarios puedan utilizar la aplicación como se espera.

Tipos de test

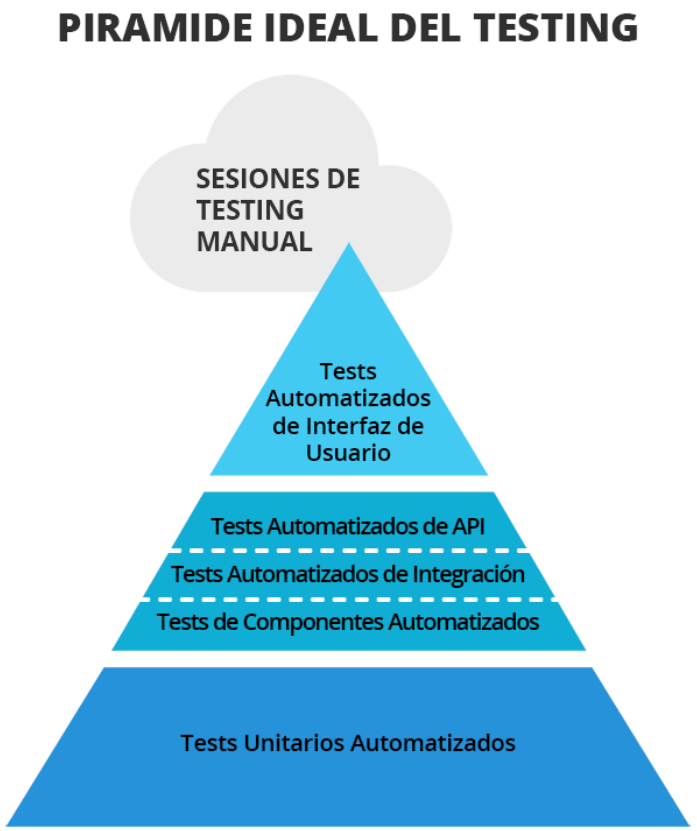
La realidad es que no existe una sola clase de verificación pero más de 150!

Different Types of Testing in Software: 100 Examples (guru99.com)

Pirámide: Costo vs Cantidad

Mayor Interacción

Mayor aislamiento



Tipos de test

Es importante considerar múltiples tipos de test en una aplicación, pero nos enfocaremos en dos específicos:

- **Tests de unidad**: Evalúan el comportamiento específico de una unidad, esto puede ser una función específica o un contexto independiente.

- **Tests de integración**: Evalúan el comportamiento de una o más unidades en conjunto!

Testing en Python

Muchas librerías permiten evaluar aplicaciones de Python en diferentes aspectos.

pytest es la más común para testing de unidad e integración.

La librería posee un sistema de plugins que permiten extender sus capacidades como sea necesario, por ejemplo para usarla en aplicaciones flask .

Testing en Python: assert

assert es una función de Python que permite evaluar que la condición pasada sea verdadera, o causará error

```
def test_siempre_pasa():
    assert True

def test_siempre_falla():
    assert False
```

Effective Python Testing With Pytest – Real Python

Pytest: requisitos

Pytest automáticamente decide que funciones son tests de acuerdo a su nombre y a su módulo.

El módulo debe empezar con **test_**
Y las funciones a evaluar también.

Este archivo será detectado

```
# contenido test_ejemplo.py
def incrementar(x):
    return x + 1


def test_incremento():
    assert incrementar(3) == 5
```

Este archivo no será detectado

```
# contenido ejemplo.py
def incrementar(x):
    return x + 1


def test_incremento():
    assert incrementar(3) == 5
```

pytest: helps you write better programs — pytest documentation

Pytest: estructura

La librería no determina ninguna estructura específica, siempre y cuando pueda descubrir los archivos de tests sin problema.

Sin embargo, dependiendo del proyecto existen varias formas de organizar los tests.

Una manera muy común, es crear una carpeta tests para todo el paquete, en la que se coloquen archivos test_ correspondientes.

pytest: ejecutar los tests

Una vez que se haya instalado la librería con Python se puede ejecutar el comando dentro de la carpeta principal del proyecto.

```
python -m pytest
```

```
============================================================ test session starts
============================================================
platform win32 -- Python 3.10.5, pytest-7.1.2, pluggy-1.0.0
rootdir: C:\Users\deort\OneDrive\Documents\GitHub\ista-python-curso-2022
collected 1 item

escuela_api\tests\test_estudiantes.py .                                  [100%]

============================================================ 1 passed in 0.04s
============================================================
```

Python:
Una herramienta multipropósito

PYTHON: CHEQUEOS

Problema

Recordemos que Python es un lenguaje dinámico, es decir que sus variables no poseen un tipo específico y el mismo puede variar durante la vida de la aplicación.

Esto puede causar problemas en el código debido a la falta de consistencia o errores lógicos creados sin intensión.

linter

Linting es en proceso de señalar errores sintácticos y estilísticos, lo que permite identificar y corregir errores programáticos o prácticas no convencionales que pueden llevar a errores.

Por ejemplo un linter detecta variables que no han sido inicializadas, llamadas a funciones no definidas, paréntesis faltantes, etc.

formatter

Un formatter es una herramienta que ajusta el formato del código para que el mismo siga una estructura consistente, o que siga convenciones especificadas de acuerdo a una configuración arbitraria.

Importancia

En conjunto, un linter y uno o más formatters pueden permitir que nuestro código sea mantenible y prevenir errores o problemas de consistencia en un entorno colaborativo.

Esto es particularmente importante cuando trabajamos en proyectos grandes cuando muchas personas pueden cambiar la estructura o el contenido del código, ya que permite estandarizar las verificaciones que se realicen en el mismo, así como su apariencia.

Importancia

Comúnmente, los IDEs o entornos de desarrollo vienen con estas capacidades por defecto. Sin embargo, existen librerías que se pueden instalar de manera similar programáticamente para realizar chequeos en entornos automatizados.

Importancia

En conjunto, un linter y uno o más formatters pueden permitir que nuestro código sea mantenible y prevenir errores o problemas de consistencia en un entorno colaborativo.

Esto es particularmente importante cuando trabajamos en proyectos grandes cuando muchas personas pueden cambiar la estructura o el contenido del código, ya que permite estandarizar las verificaciones que se realicen en el mismo, así como su apariencia.

Flake8

pyflakes: Un linter muy simple que chequea archivos Python. La herramienta analiza errores y malas prácticas de código. Flake8 es una herramienta que ejecuta pyflakes y otra serie de scripts útiles.

```
python3 -m pip install flake8
flake8 ubicacion/del/codigo
```

flake8 · PyPI

black

Una herramienta opinionada que formatea el código de acuerdo a un estilo único y consistente, cualquier código que usa black será formateado de la misma manera independientemente de su contenido.

```
python -m pip install black
black  ubicacion/del/codigo
```

psf/black: The uncompromising Python code formatter (github.com)

Automatización de chequeos

Una plataforma de integración y emisión continua permite al desarrollador ejecutar acciones automatizadas que aplican al proyecto.

Integración continua es la práctica de automatizar la integración de los cambios de código en un proyecto de software, esto significa chequeos, pruebas y testing.

¡Existen ciertas plataformas gratuitas disponibles, que permiten integrar automáticamente estos beneficios a nuestros repositorios de Github!

Features • GitHub Actions
CircleCI vs GitHub Actions - CircleCI

¿En qué consiste la integración continua? | Atlassian