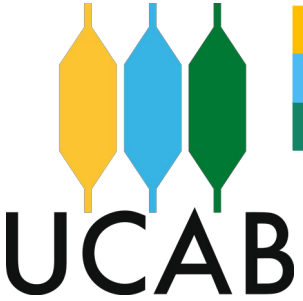


Programación de APIs con Python

Decoradores Ambientes Virtuales Pydantic

Profesor:
José Gregorio Castillo Pacheco

Decoradores – Ambientes Virtuales

Temario

- Decoradores
- Ambientes
- Pydantic

Decoradores

- Antes de hablar de los decoradores: ¿Qué son los patrones de diseño?
- El concepto de patrones fue adoptado por Erich Gamma, John Vlissides, Ralph Johnson y Richard Helm, quienes publicaron "Patrones de diseño" en 1995, aplicando el concepto de patrones de diseño a la programación.
- El libro presentaba 23 patrones que resolvían varios problemas del diseño orientado a objetos
- Desde entonces, se han descubierto muchos otros patrones orientados a objetos y la metodología del patrón se ha hecho popular en otros campos de la programación.
- Los patrones de diseño son soluciones habituales a problemas comunes en el diseño orientado a objetos.

Decoradores

- Los patrones de diseño se clasifican según su propósito y se pueden dividir en tres categorías:
  - Los **patrones creacionales** proporcionan mecanismos de creación de objetos que incrementan la flexibilidad y la reutilización de código existente.
  - Los **patrones estructurales** explican cómo ensamblar objetos y clases en estructuras más grandes a la vez que se mantiene la flexibilidad y eficiencia de la estructura.
  - Los **patrones de comportamiento** se encargan de una comunicación efectiva y la asignación de responsabilidades entre objetos.

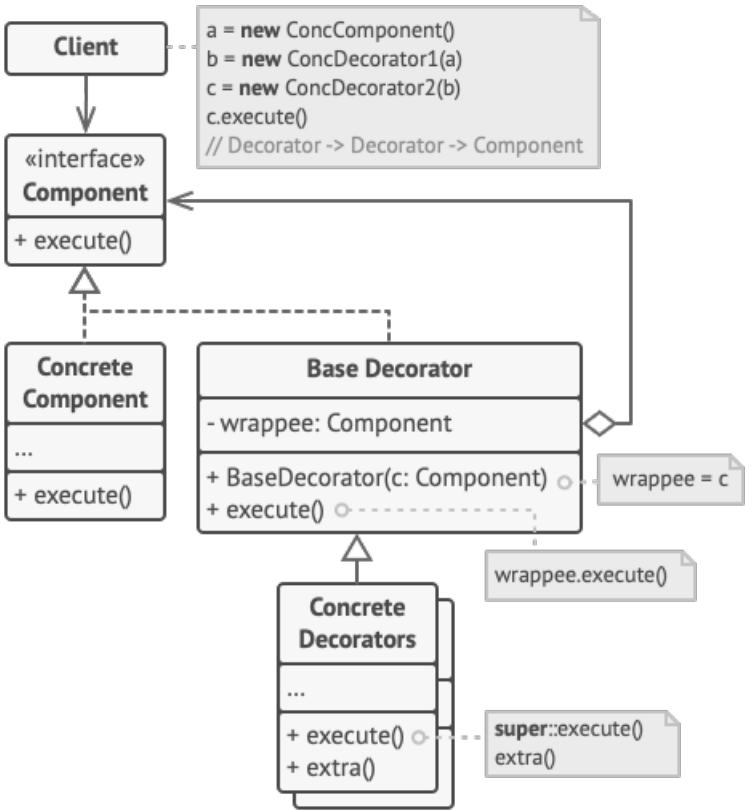
Decoradores

- El patrón **Decorador:**
  - Es un patrón de diseño de tipo estructural que permite añadir funcionalidades a objetos colocando estos objetos dentro de objetos encapsuladores especiales que contienen estas funcionalidades.
  - Se basa en los siguientes elementos:
    - Componente: El objeto que se va a decorar.
    - Decorador: El objeto que proporciona las funcionalidades adicionales.
    - Interfaz: La interfaz que implementan tanto el componente como el decorador.

Decoradores

- El patrón **Decorador:**

1. El **Component** declara la interfaz común tanto para wrappers como para objetos envueltos.
2. **Concrete Component** es una clase de objetos envueltos. Define el comportamiento básico, que los decoradores pueden alterar.
3. La clase **Base Decorador** tiene un campo para referenciar un objeto envuelto. El tipo del campo debe declararse como la interfaz del componente para que pueda contener tanto los componentes concretos como los decoradores. La clase decoradora (base decorator) base delega todas las operaciones al objeto envuelto.
4. Los **Concrete Decorator** definen funcionalidades adicionales que se pueden añadir dinámicamente a los componentes. Los decoradores concretos sobrescriben métodos de la clase decoradora base y ejecutan su comportamiento, ya sea antes o después de invocar al método padre.



Ver ejemplo: simple_decorator

Referencia: https://refactoring.guru/es/design-patterns/decorator

Decoradores

- El patrón **Decorador:** lo visto anteriormente, es implementado de acuerdo a lo sugerido por el GanOfFour
- Python lo hace aún más sencillo, pero antes debemos manejar algunos conceptos
  - Python permite **asignar funciones a una variable**:

```
def plus_one(number):
    return number + 1

add_one = plus_one
add_one(5)

6
```

Ver ejemplo: ej_2_variable_como_funcion.py

Decoradores

- **Funciones anidadas**:
- Python permite definir una función dentro de otra función:

```
def plus_one(number):
    def add_one(number):
        return number + 1

    result = add_one(number)
    return result


plus_one(4)

5
```

Ver ejemplo: ej_3_funcion_interna.py

Decoradores

- **Pase de función como argumento**:
- Python permite pasar una función como argumento de otra función:

```
def plus_one(number):
    return number + 1

def function_call(function):
    number_to_add = 5
    return function(number_to_add)

function_call(plus_one)

6
```

Ver ejemplo: ej_4_funcion_como_argumento.py

Decoradores

- **Función como retorno**:
- Python permite usar una función como retorno de otra función:

```
def hello_function():
    def say_hi():
        return "Hi"
    return say_hi

hello = hello_function()
hello()

'Hi'
```

Ver ejemplo: ej_5_funcion_como_retorno.py

Decoradores

- **Acceso a las variables de la Función Anexa, por la Función anidada**:
  - Python permite a una función anidada acceder al ámbito externo de la función que la encierra. Este es un concepto crítico en los decoradores -- este patrón se conoce como **Closure**.

```
def print_message(message):
    """Enclosing Function"""
    def message_sender():
        """Nested Function"""
        print(message)

    message_sender()

print_message("Some random message")
```

Ver ejemplo: ej_6_acceso_interno.py

Decoradores

- **Decoradores**:
- Con estos requisitos previos superados, vamos a crear un decorador sencillo que convierta una sentencia a mayúsculas.
- Hacemos esto definiendo un wrapper dentro de una función contenida.
- Como puedes ver es muy similar a la función dentro de otra función que creamos anteriormente.

```
def uppercase_decorator(function):
    def wrapper():
        func = function()
        make_uppercase = func.upper()
        return make_uppercase

    return wrapper
```

```
def say_hi():
    return 'hello there'

decorate = uppercase_decorator(say_hi)
decorate()


'HELLO THERE'
```

Decoradores

- **Decoradores**:
- Python nos proporciona una forma mucho más sencilla de aplicar decoradores.
- Simplemente usamos el símbolo @ antes de la función que queremos decorar.

```
def uppercase_decorator(function):
    def wrapper():
        func = function()
        make_uppercase = func.upper()
        return make_uppercase

    return wrapper
```

```
@uppercase_decorator
def say_hi():
    return 'hello there'

say_hi()


'HELLO THERE'
```

Ver ejemplo: ej_7_uppercase_decorator.py

Decoradores

- **Decoradores**:
- Python permite usar varios decoradores en una misma función.
- Los decoradores se aplicarán en el orden en que se han llamado

```
def split_string(function):
    def wrapper():
        func = function()
        splitted_string = func.split()
        return splitted_string

    return wrapper
```

```
@split_string
@uppercase_decorator
def say_hi():
    return 'hello there'

say_hi()

['HELLO', 'THERE']
```

- Notamos que la aplicación de los decoradores es de abajo hacia arriba.
- Si hubiéramos intercambiado el orden, habríamos visto un error, ya que las listas no tienen atributo upper. La frase se ha convertido primero a mayúsculas y luego se ha dividido en una lista.

Ver ejemplo: ej_8_several_decorators.py

Decoradores

- **Decoradores**:
- Los decoradores pueden aceptar argumentos.
- Esto se consigue pasando los argumentos a la función wrapper. Los argumentos se pasarán a la función que se está decorando en el momento de la llamada.

```
def decorator_with_arguments(function):
    def wrapper_accepting_arguments(arg1, arg2):
        print(f"My arguments are: {arg1}, {arg2}")
        function(arg1, arg2)
    return wrapper_accepting_arguments

@decorator_with_arguments
def cities(city_one, city_two):
    print(f"Cities I love are {city_one} and {city_two}")
```

Ver ejemplo: ej_9_decorator_con_argumentos.py

Decoradores

- **Decoradores**:
- Los decoradores pueden ser de propósito general.
- Simplemente se utiliza *args y **kwards como los parámetros a usar

```
def a_decorator_passing_arbitrary_arguments(function_to_decorate):
    def a_wrapper_accepting_arbitrary_arguments(*args,**kwargs):
        print('los argumentos posicionales son', args)
        print('Los argumentos de clave son', kwargs)
        function_to_decorate(*args, **kwargs)
    return a_wrapper_accepting_arbitrary_arguments
```

Ver ejemplo: ej_10_argumentos_arbitrarios.py

Decoradores

- **Decoradores**:
- También podemos pasar argumentos al decorador
- Para ello, definimos un creador de decoradores que acepte argumentos y, a continuación, definimos un decorador dentro de él.

```
def decorator_maker_with_arguments(decorator_arg1, decorator_arg2, decorator_arg3):
    def decorator(func):
        def wrapper(function_arg1, function_arg2, function_arg3) :
            "This is the wrapper function"
            print("El wrapper puede acceder a todas la variables\n"
                  "\t- desde el decorator: {0} {1} {2}\n"
                  "\t- desde la llamada a la function: {3} {4} {5}\n"
                  "y pasarcelas a la función decorada"
                  .format(decorator_arg1, decorator_arg2, decorator_arg3,
                          function_arg1, function_arg2,function_arg3))
            return func(function_arg1, function_arg2,function_arg3)

        return wrapper

    return decorator
```

Ver ejemplo: ej_11_decorador_con_argumentos.py

Decoradores

- **Decoradores**:
- Python nos proporciona una herramienta para depurar los decoradores
- En el paquete **functools** está el factory **wraps**

```
from functools import wraps

def my_decorator(function):
    @wraps(function)
    def wrapper(*args, **kwards):
        print("Llamando a la función decorada")
        return function(*args, **kwards)
    return wrapper
```

```
@my_decorator
def example():
    """Docstring"""
    print("Llamada a la función
example")
```

- Sin el uso de esta fábrica de decoradores, el nombre de la función de ejemplo habría sido 'wrapper', y se habría perdido el docstring del ejemplo() original.

Ver ejemplo: ej_12_using_wraps.py

Ambientes Virtuales

- Un entorno virtual Python es un directorio autónomo que contiene su propio intérprete y bibliotecas Python, separado de la instalación Python de todo el sistema.
- Se utiliza para crear un espacio aislado donde se pueden instalar paquetes específicos para un proyecto sin afectar la instalación principal de Python en el sistema.
- Esto permite evitar conflictos de paquetes entre diferentes proyectos, y también facilita la administración de las dependencias de cada proyecto.

Ambientes Virtuales

Importancia de usar ambientes virtuales en el desarrollo de software en Python:

1. **Gestión de Dependencias**: Los proyectos de Python a menudo requieren diferentes versiones de bibliotecas o paquetes. Los ambientes virtuales ayudan a gestionar estas dependencias de manera aislada para evitar conflictos.
2. **Evitar Contaminación**: Sin ambientes virtuales, los paquetes instalados globalmente en Python pueden entrar en conflicto entre sí o con el sistema operativo, lo que puede causar problemas de compatibilidad.
3. **Portabilidad**: Los ambientes virtuales hacen que los proyectos sean más portátiles. Es posible compartir el ambiente virtual con otros desarrolladores, lo que garantiza que todos tengan la misma configuración.
4. **Mantenimiento**: Los ambientes virtuales facilitan la gestión a largo plazo de proyectos, ya que cada proyecto se puede aislar y mantener por separado.

Una práctica común, y recomendable, cuando se desarrolla en Python es hacer uso de ambientes virtuales.

Ambientes Virtuales

- Creación de un ambiente virtual:
  - Simplemente se ejecuta el siguiente comando:

```
$ python -m venv venv
```

o

```
$ python3 -m venv venv
```

(dependiendo de nuestra instalación)

- la primera palabra invoca al interpretador de Python,
- la **-m** es un parámetro para indicarle al interpretador que va a ejecutar un comando, ese comando es **venv** el cual es el encargado de generar una carpeta donde estará un interpretador de Python y las librerías básicas para su funcionamiento, incluido pip, para descargar las dependencias que hagan falta para el proyecto.
- Por último está el nombre que le daremos a nuestro ambiente y esta puede ser cualquier palabra que deseemos, además de ser el nombre de la carpeta donde residirá este ambiente, en este caso es **venv**

Ambientes Virtuales

- **Activación del ambiente virtual**
- Dependiendo del sistema operativo, podemos activar el ambiente virtual usando alguno de los siguientes comandos:

Windows (usando el Command Prompt):

```
$ venv\Scripts\activate
```

Windows (usando PowerShell):

```
$ .\venv\Scripts\activate.ps1
```

macOs y Linux:

```
$ source venv/bin/activate
```

- Donde `venv` es el nombre de la carpeta que usamos cuando creamos el ambiente virtual.
- Una vez activado el ambiente virtual, el prompt del terminal lo indicará anteponiendo al simbolo del mismo el nombre de la carpeta que le indicamos.

Ambientes Virtuales

- **Desactivando el ambiente virtual**
- Para salir del ambiente virtual y retornar al ambiente de Python del sistema, simplemente se ejecuta el siguiente comando:

```
$ deactivate
```

- **Ejemplos prácticos de ambientes virtuales (práctica)**

Ambientes Virtuales

- PIP es un manejador de paquetes de Python, o módulos si prefiere llamarlo así
- Es posible instalar, actualizar y quitar paquetes usando **pip**.
- Por defecto pip instalará paquetes desde el **P**ython **P**ackage **I**ndex (Indice de Paquetes Python), <https://pypi.python.org/pypi> .
- Se puede navegar el Python Package Index ingresando con el navegador de internet, o se puede usar la búsqueda limitada de pip‘s

Ambientes Virtuales

```
(tutorial-env) $ pip search astronomy
skyfield            - Elegant astronomy for Python
gary                - Galactic astronomy and gravitational dynamics.
novas               - The United States Naval Observatory NOVAS astronomy library
astroobs            - Provides astronomy ephemeris to plan telescope observations
PyAstronomy         - A collection of astronomy related tools for Python.
…
```

Ambientes Virtuales

- pip tiene varios subcomandos: “search”, “install”, “uninstall”, “freeze”, etc. (consulte la guía Instalando módulos de Python para la documentación completa de pip.)
- Se puede instalar la última versión de un paquete especificando el nombre del paquete:

```
(tutorial-env) $ pip install novas
Collecting novas
  Downloading novas-3.1.1.3.tar.gz (136kB)
Installing collected packages: novas
  Running setup.py install for novas
Successfully installed novas-3.1.1.3
```

Ambientes Virtuales

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

```
(tutorial-env) $ pip install requests==2.6.0
Collecting requests==2.6.0
  Using cached requests-2.6.0-py2.py3-none-any.whl
Installing collected packages: requests
Successfully installed requests-2.6.0
```

Ambientes Virtuales

Si se re-ejecuta 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:

```
(tutorial-env) $ pip install --upgrade requests
Collecting requests
Installing collected packages: requests
  Found existing installation: requests 2.6.0
    Uninstalling requests-2.6.0:
      Successfully uninstalled requests-2.6.0
Successfully installed requests-2.7.0
```

Ambientes Virtuales

pip uninstall seguido de uno o varios nombres de paquetes desinstalará los paquetes del entorno virtual.

pip show mostrará información de un paquete en particular:

```
(tutorial-env) $ pip show requests
---
Metadata-Version: 2.0
Name: requests
Version: 2.7.0
Summary: Python HTTP for Humans.
Home-page: http://python-requests.org
Author: Kenneth Reitz
Author-email: me@kennethreitz.com
License: Apache 2.0
Location: /Users/akuchling/envs/tutorial-env/lib/python3.4/site-packages
Requires:
```

Ambientes Virtuales

pip list mostrará todos los paquetes instalados en el entorno virtual**:**

```
(tutorial-env) $ pip list
novas (3.1.1.3)
numpy (1.9.2)
pip (7.0.3)
requests (2.7.0)
setuptools (16.0)
```

Ambientes Virtuales

pip freeze devuelve una lista de paquetes instalados similar, pero usa el formato de salida requerido por **pip install**.

Una convención común es poner esta lista en un archivo **requirements.txt**::

```
(tutorial-env) $ pip freeze > requirements.txt
(tutorial-env) $ cat requirements.txt
novas==3.1.1.3
numpy==1.9.2
requests==2.7.0
```

Ambientes Virtuales

El archivo **requirements.txt** entonces 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:

```
(tutorial-env) $ pip install -r requirements.txt
Collecting novas==3.1.1.3 (from -r requirements.txt (line 1))
  ...
Collecting numpy==1.9.2 (from -r requirements.txt (line 2))
  ...
Collecting requests==2.7.0 (from -r requirements.txt (line 3))
  ...
Installing collected packages: novas, numpy, requests
  Running setup.py install for novas
Successfully installed novas-3.1.1.3 numpy-1.9.2 requests-2.7.0
```

Ambientes Virtuales

Si buscan proyectos de Python en github, se encontrarán que algunos no usan pip, sino que usan un paquete llamado *Poetry*

Poetry es un gestor de paquetes moderno y completo para Python que simplifica enormemente la gestión de dependencias en los proyectos.

A diferencia de pip, el gestor de paquetes estándar de Python, Poetry ofrece un enfoque más estructurado y unificado para manejar todas las tareas relacionadas con los paquetes, desde la instalación y actualización de dependencias hasta la creación de entornos virtuales aislados.

Ambientes Virtuales

¿Cuáles son las principales ventajas de Poetry sobre pip?

- **Gestión de dependencias más robusta**: Poetry utiliza un archivo *pyproject.toml* para definir todas las dependencias del proyecto, lo que facilita la resolución de conflictos de versiones y garantiza la reproducibilidad del entorno de desarrollo. Además, Poetry permite especificar dependencias de desarrollo y crear entornos virtuales de forma más intuitiva.

- **Flujo de trabajo más eficiente**: Con Poetry, se puede crear y activar entornos virtuales con un solo comando, instalar y actualizar paquetes de forma rápida y sencilla, y generar archivos de requisitos para compartir el proyecto con otros. Esto agiliza significativamente el desarrollo y reduce la probabilidad de errores.

- **Mejor integración con otras herramientas**: Poetry se integra de forma nativa con muchas herramientas de desarrollo populares, como linters, formateadores y herramientas de testing, lo que te permite crear un flujo de trabajo de desarrollo más coherente y eficiente.

FIN