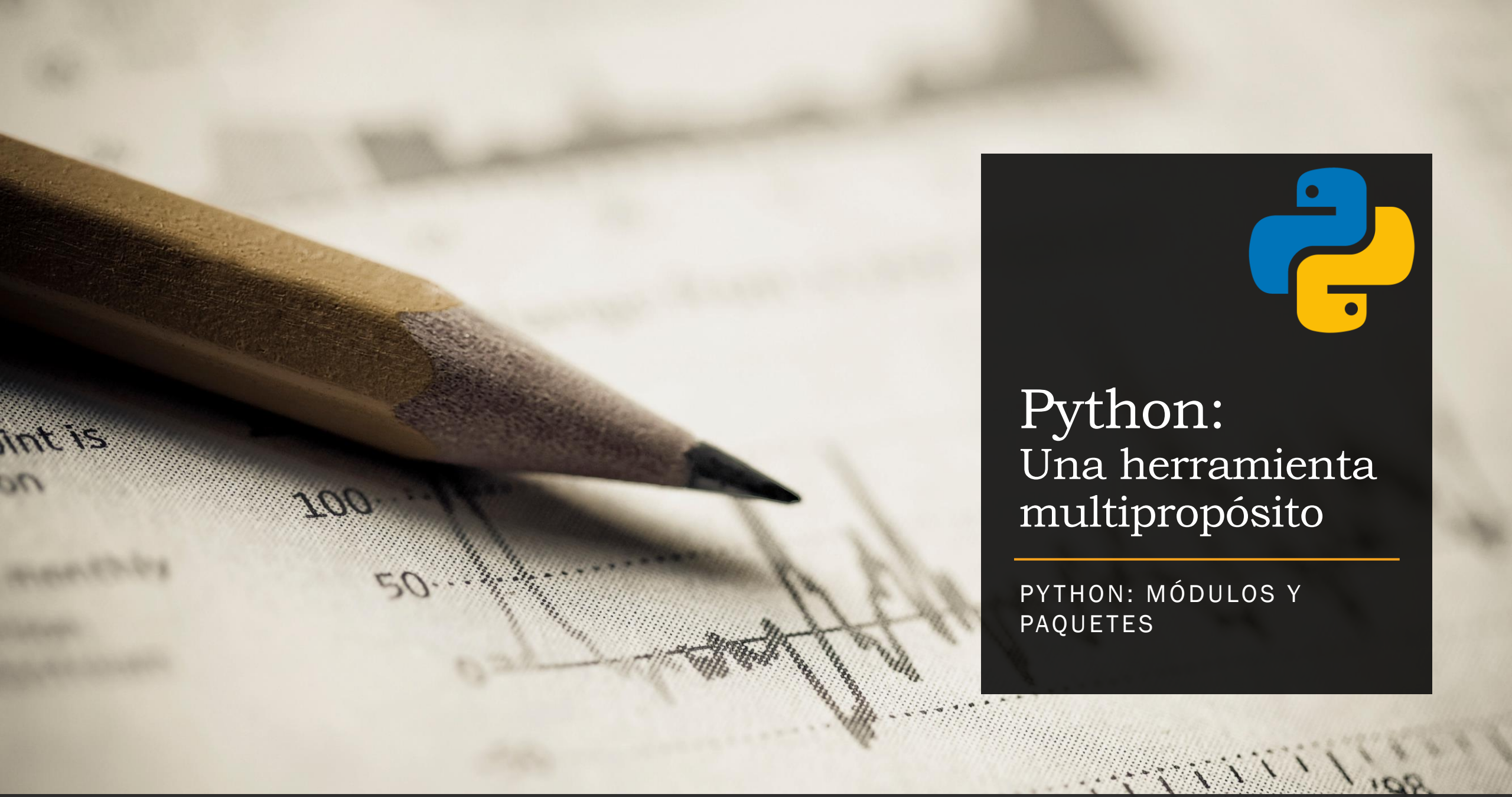


Python:
Una herramienta multipropósito

PYTHON: MÓDULOS Y PAQUETES

Esta clase

- Módulos, un poco más de detalle
- Estructura de proyectos
- Gestión de dependencias
- Tipos
- Introducción a sistemas de control de versiones
- Creando un repositorio en Github

Recordando

Un módulo sencillo

```python
class Persona:
    # El constructor siempre se define así
    # El argumento self siempre debe estar presente
    def __init__(self, nombre, cedula):
        # Atributos se definen en la instancia self
        self.nombre = nombre
        self.cedula = cedula

    # El argumento self siempre debe estar presente ya que representa al objeto
    def imprimir_datos(self):
        print(f"{self.nombre} tiene cedula {self.cedula}")



# Creamos objetos para esta clase
a = Persona("Daniel", 12134142)
b = Persona("David", 454545)
a.imprimir_datos()
b.imprimir_datos()
```

¿Qué es un módulo?

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.

¿Qué es un módulo?

Un módulo esencialmente es un archivo con código Python que contiene definiciones de funciones y código.

Las definiciones de un módulo pueden ser utilizadas en diferentes módulos, o dentro del intérprete como veamos necesario.

Recordemos: Un archivo en Python suele terminar con la extensión **.py**

¿Qué es un módulo?

Python identifica a un módulo mediante el nombre de su archivo.

Por ejemplo, un archivo mi_modulo.py podrá ser importado mediante mi_modulo

mi_modulo.py

```
def suma(a, b):
    return a + b

def multiplicacion(a, b):
    return a * b
```

otro_modulo.py

```
import mi_modulo

print(mi_modulo.suma(5, 8)) #13
```

O desde el intérprete interactivo

```
>>> import mi_modulo
>>> mi_modulo.suma(1000)
0
```

¿Qué es un módulo?

Alternativamente, podemos importar las funciones directamente

mi_modulo.py

```python
def suma(a, b):
    return a + b

def multiplicacion(a, b):
    return a * b
```

otro_modulo.py

```python
from mi_modulo import suma

print(suma(5, 8)) #13
```

Sin embargo, esto tiene implicaciones en cuanto a los espacios de nombre, ya que pueden existir varias funciones suma dentro del contexto actual.

Espacios de nombres

Un espacio de nombre es simplemente una relación entre los nombres de las variables y los objetos a los que ellos identifican. En otras palabras determinan un contexto común.

Un módulo en general determina un espacio de nombre para todas las definiciones dentro mismo, y es por ello que varios módulos pueden sin problema especificar funciones o atributos con el mismo nombre.

mi_modulo_1.py

```
def mutiplicar_por_tres(a):
    return a * 3
```

mi_modulo_2.py

```
def mutiplicar_por_tres(a):
    return a + a + a
```

otro_modulo.py

```
# Incluimos los otros módulos en el espacio de nombres actual
import mi_modulo_1
import mi_modulo_2

print(mi_modulo_1.mutiplicar_por_tres(5)) #15
print(mi_modulo_2.mutiplicar_por_tres(5)) #15
```

Espacios de nombres

En otras palabras, al importar algo lo estamos incluyendo en el espacio de nombres en el que se ejecuta el código.

Pero, ¿Qué sucede si agregamos más de un objecto con el mismo identificador al espacio?

```
from mi_modulo_1 import mutiplicar_por_tres
from mi_modulo_2 import mutiplicar_por_tres

# Funciona pero ambas llamadas usarán la función
# de mi_modulo_2 ya que fue evaluada más recientemente
print(mutiplicar_por_tres(5)) #15
print(mutiplicar_por_tres(5)) #15
```

Espacios de nombres:

Entonces, un espacio de nombre puede tener **un solo objeto** con el mismo identificador.

En ese caso podemos siempre usar aliases/seudónimos/apodos, como queramos llamarles.

Simplemente reemplaza el identificador que agregamos al espacio de nombre, usando la palabra **as**

```
# Un alias para diferenciarlas al inyectarlas en el espacio de nombres
from mi_modulo_1 import mutiplicar_por_tres as mutiplicar_por_tres_1
from mi_modulo_2 import mutiplicar_por_tres as mutiplicar_por_tres_2

# Funciona y ambas usaran la función que queremos
print(mutiplicar_por_tres_1(5)) #15
print(mutiplicar_por_tres_2(5)) #15
```

Librería estándar

Entonces un módulo puede importar módulos de manera muy sencilla, ya sean estos en la misma ubicación en el disco, o módulos base que vienen de fábrica en Python.

Algunos módulos de fábrica

- datetime: operar con fechas y tiempos
- hashlib: operaciones criptográficas
- itertools: herramientas para crear iteradores
- os: herramientas varias para controlar el sistema operativo

Python Module Index — Python 3.10.5 documentation

Continuando

Un módulo puede contener definiciones de funciones que pueden ser importadas por otros módulos, así como declaraciones ejecutables que se ejecutan por única vez al momento de llamar al módulo con el intérprete o al importarlo mediante **import** en otro lugar.

Como regla general, si el código en un script no está protegido por un condicional, o si no es una función, siempre se ejecutará cuando se importe.

Python Module Index — Python 3.10.5 documentation

Importando

Como vimos en la clase anterior, Python requiere que el código exista solamente al momento en el que se ejecuta, es decir podemos importar cosas en cualquier ubicación.

Sin embargo, por simpleza y para mantener nuestro código organizado, es mejor siempre colocar todos las dependencias en la parte superior del módulo.

```
print("Hola")
import mi_modulo_1
print(mi_modulo_1.mutiplicar_por_tres(5)) #15
```

✗

```
import mi_modulo_1
print("Hola")
print(mi_modulo_1.mutiplicar_por_tres(5)) #15
```

✓

Importando

Adicionalmente es posible importar todos los atributos del un módulo al espacio de nombres actual, sin embargo. Esto está muy mal visto por varias razones:

- Se desconoce de donde provienen las funciones al leer el código
- No es obvio lo que se usa en realidad
- Incluye funciones que no son necesariamente públicas
- Puede reemplazar funciones con nombres iguales (recuerden, no puede existir más de una cosa con ese nombre).

```
from mi_modulo_1 import *
print(mutiplicar_por_tres(5)) #15_
```

✕

Módulos como scripts

Un script es nada más un archivo que se ejecuta. Un módulo puede ejecutarse simplemente al llamarlo mediante el intérprete como se ha demostrado previamente.

Todos los módulos son naturalmente scripts ya que pueden invocarse directamente.

modulo_ejemplo.py

```
def mi_funcion(a, b):
    print(a, b)

print("Hola mi nombre es modulo_ejemplo")
```

Módulos como scripts

Como vimos antes, Python ejecuta el código que un módulo declara cuando se lo usa como script o cuando se lo importa desde otro módulo.

modulo_ejemplo.py

```
def mi_funcion(a, b):
    print(a + b)

print("Hola mi nombre es modulo_ejemplo")
```

Importado

otro_modulo.py

```
from modulo_ejemplo importar mi_funcion
mi_funcion(1, 2)
```

```
>> python3 otro_modulo.py
Hola mi nombre es modulo_ejemplo
3
```

Ejecutado como script

```
>> python3 mi_modulo.py
Hola mi nombre es modulo_ejemplo
```

Módulos como scripts

Entonces ¿cómo podemos controlar la ejecución de código dentro de un módulo, dependiendo de como se usa?

- ¿Puedo ejecutar cierto código solo cuando se usa como script?
- ¿Puedo ejecutar cierto código solo cuando se importa?

Sí, usemos la variable **__name__** !

__name__

Variable “mágica” disponible en cualquier módulo por defecto.

Python la define de dos maneras diferentes.

- Cuando el módulo es el módulo principal, es decir, se invocó desde el intérprete o como script, su valor es “__main__”
- Cuando el módulo es importado, su valor es el nombre asignado por el módulo que lo importó.

Es decir, tenemos mucha flexibilidad.

__name__

modulo_ejemplo.py

```python
def mi_funcion(a, b):
    print(a + b)

print(__name__)
```

Ejecutado como script

```
>> python3 modulo_ejemplo.py
__main__
```

Importado

otro_modulo.py

```python
from modulo_ejemplo importar mi_funcion
mi_funcion(1, 2)
```

```
>> python3 otro_modulo.py
modulo_ejemplo
3
```

Python: Encuesta

¿Cuál será el valor que se imprimirá en la consola cuando ejecutemos este código?

utilidades.py

```
def extraer_primeras_5_letras(texto):
    return texto[0:5]
print(__name__)
```

mi_programa.py

```
from utilidades import extraer_primeras_5_letras
print(extraer_primeras_5_letras("abcdefgh"))
print(__name__)
```

```
doc:~$ python3  mi_programa.py
?
?
?
```

La encuesta se abrirá en Zoom

Python: Encuesta

¿Cuál será el valor que se imprimirá en la consola cuando ejecutemos este código?

utilidades.py

```python
def extraer_primeras_5_letras(texto):
    return texto[0:5]
print(__name__)
```

mi_programa.py

```python
from utilidades import extraer_primeras_5_letras
print(extraer_primeras_5_letras("abcdefgh"))
print(__name__)
```

```
doc:~$ python3  mi_programa.py
utilidades
abcde
__main__
```

La encuesta se abrirá en Zoom

__name__

Entonces, podemos usar __name__ para controlar la ejecución de código dependiendo de como el módulo sea utilizado.

```
def mi_funcion(a, b):
    print(a, b)

# Este código solo correrá cuando el módulo se ejecute
directamente
# NO cuando se importe.
if __name__ == "__main__":
    print("Hola me estás ejecutando como script")
```

dir

La función integrada dir() permite determinar los atributos que expone un módulo, por ejemplo

```
import mi_modulo
dir(mi_modulo)
>>> [‘__name__’, mi_funcion]
```

Encontrando módulos

¿Cuándo decimos a un módulo que importe a otro, cómo sabe Python dónde encontrarlo?

Existe un camino de búsqueda que por defecto el intérprete usa.

1. Inicialmente busca el módulo importado en la librería de fábrica de Python
2. Continúa buscando por un archivo con el nombre del módulo en la variable **sys.path**

La variable **sys.path** se inicializa con los siguientes directorios:

1. El directorio que contiene el script inicial, es decir de donde se inició el intérprete.
2. La variable del sistema PYTHONPATH
3. El directorio predeterminado de la instalación

Encontrando módulos

Por lo tanto es importante asegurarnos que el módulo que queremos esté en el lugar adecuado.

Podemos adicionar directorios cambiando la variable **sys.path**, que es simplemente una lista.

Estructura de carpeta

Generalmente cuando trabajamos en un proyecto grande, no se opera con un solo archivo.

De hecho a veces tenemos muchos más archivos agrupados en carpetas para ofrecer una mejor forma de gestionar nuestro proyecto a las personas que trabajen en él.

```
.
├── Pipfile
├── Pipfile.lock
├── operations
│   ├── grayscale
│   │   └── apply.py
│   ├── resize
│   │   └── apply.py
│   └── styletransfer
│       ├── README.md
│       ├── apply.py
│       ├── models
│       │   └── 21styles.model
│       ├── net.py
│       ├── run.py
│       └── utils.py
└── pipeline.py
```

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
```

__init__.py

Un archivo obligatorio cuando se quiere tener un paquete. Esto permite a Python identificar un directorio como una jerarquía que debe tratarse como tal.

El archivo puede estar vacío, pero también puede definir código que querramos cuando un paquete o subpaquete se importe.

Python: Encuesta

¿Cuál es la manera correcta de importar el módulo **estudiante** en este paquete?

```
escuela
├── __init__.py
├── entidades
│   ├── __init__.py
│   ├── estudiante.py
│   └── profesor.py
└── utilidades
    ├── __init__.py
    └── reportes.py
```

La encuesta se abrirá en Zoom

Conclusión: Módulos y paquetes

- En Python un módulo es un archivo que puede ser importado por otros módulos o ser ejecutado.
- Un paquete es una manera de agrupar módulos en una jerarquía que nos permite definir un espacio de nombres estructurado.

Dependencias y paquetes externos

No siempre es buena idea crear todo desde cero, la mayor parte de las veces nuestro proyecto requiere dependencias externas que ofrecen servicios que alguien más ha creado con diferentes propósitos.

Por ejemplo, para analizar datos, para conectarse a servidores de manera más eficiente, para manipular imágenes, etc.

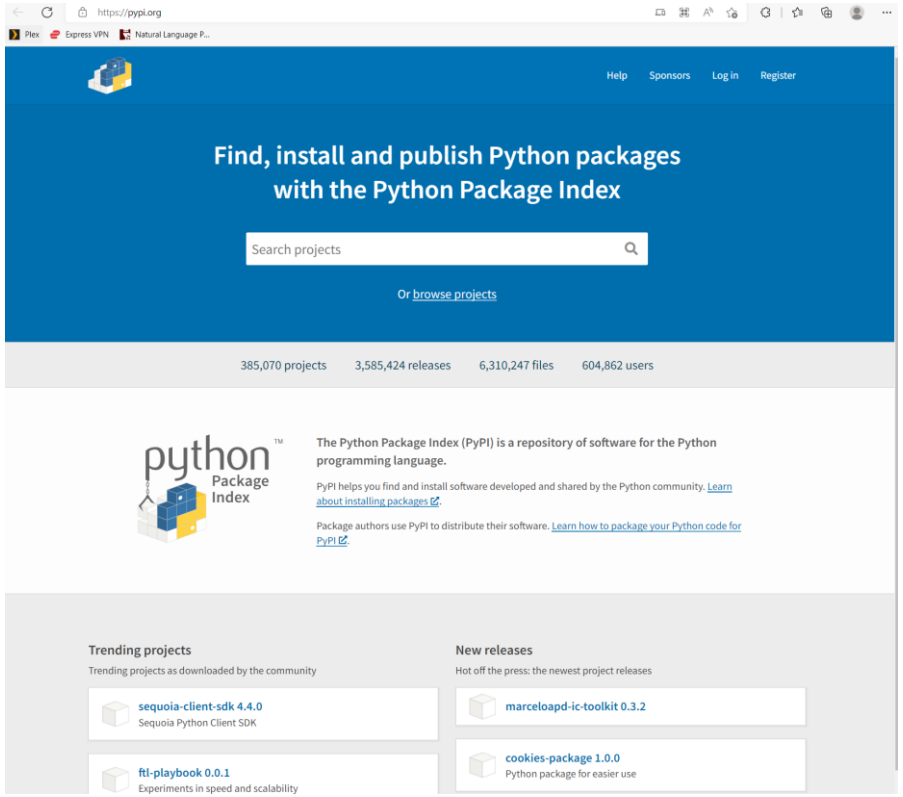
Dependencias en Python: pip

Al ser un proyecto popular de código abierto existe una amplia comunidad de soporte y contribuciones en muchos proyectos.

pip es el instalador de paquetes por defecto que viene en las últimas versiones de Python.

El **índice de paquetes de Python** es un repositorio público de proyectos de acceso libre donde se encuentran muchos paquetes (más de 300000)!

Dependencias en Python



El índice público de paquetes (PyPI)

En el repositorio público uno puede buscar proyectos de acuerdo a los requisitos que tengamos.

[PyPI · The Python Package Index]

Dependencias en Python

Cómo aprendimos anteriormente, Python descubre la ubicación de los módulos recorriendo una serie de directorios en secuencia.

Es decir, que cuando instalamos un módulo externo tenemos que asegurarnos que se encuentre en una ubicación disponible para el proyecto.

Dependencias en Python

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

Installing Python Modules — Python 3.10.5 documentation

Dependencias en Python

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.

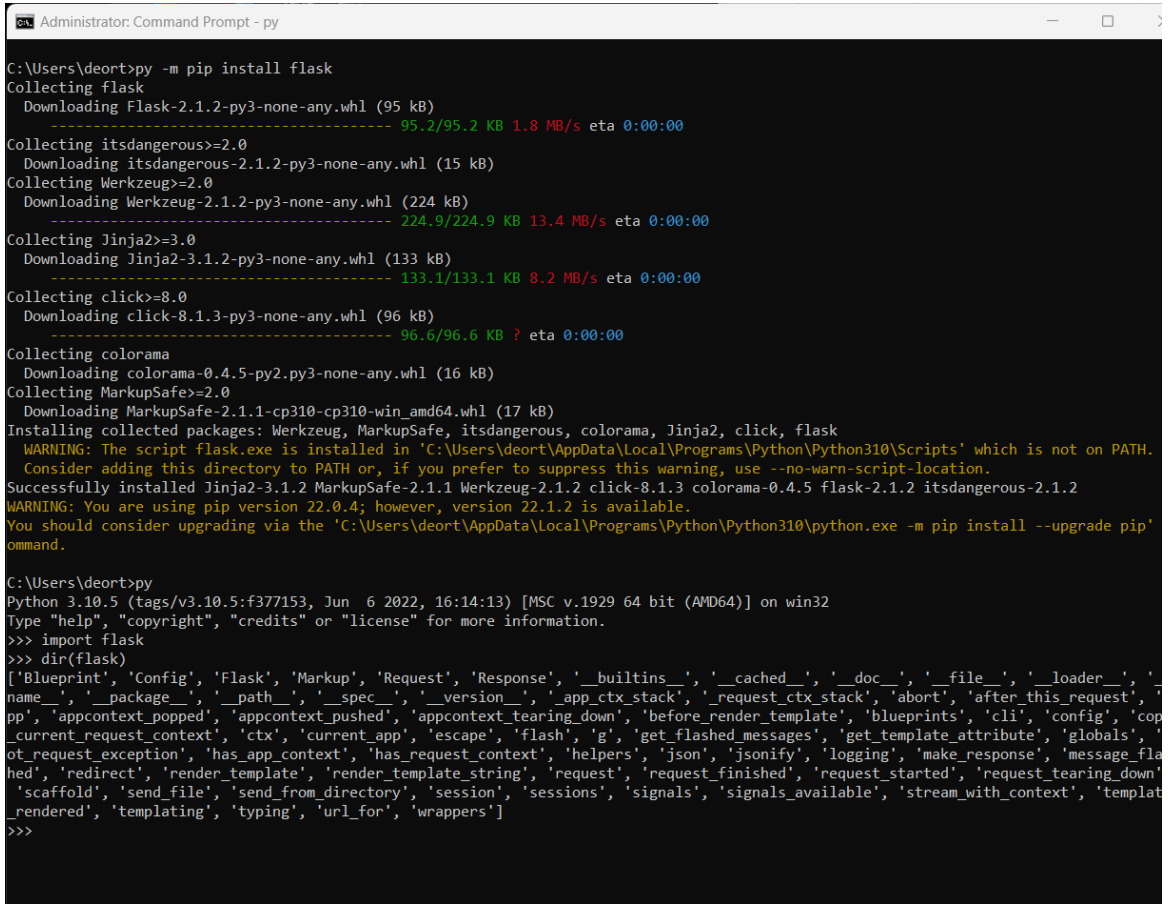
python -m pip install AlgunPaquete==1.0.4

Nota: Dependiendo de como se instaló en Windows, es posible que en lugar de **python** uno deba usar **py** desde la consola del sistema.

Installing Python Modules — Python 3.10.5 documentation

Dependencias en Python

Una vez instalada la dependencia, se puede acceder mediante **import** tal como lo hacemos con módulos locales



Dependencias en Python

En la siguiente clase aprenderemos de entornos aislados para gestionar dependencias que no se instalen a nivel global, sino que pertenezcan a un proyecto específico.

Pero en caso de tener interés, leer más sobre los entornos virtuales.

venv — Creation of virtual environments — Python 3.10.5 documentation

Conclusión

Preguntas?