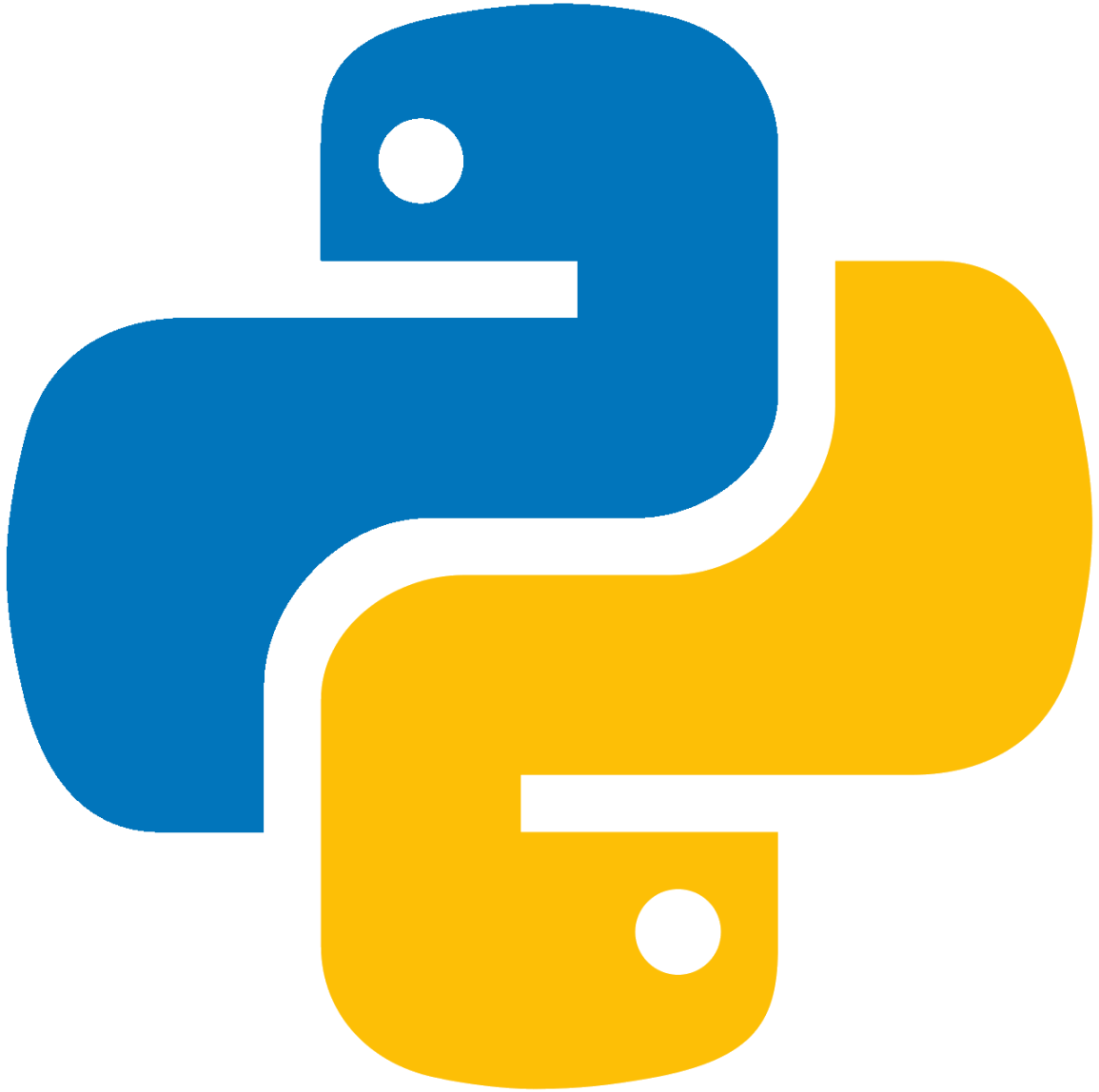


Python: Análisis de Datos y Ecosistemas Modernos de Analítica

MODULOS, PAQUETES Y TIPADO

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?

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 interactivo 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 (namespaces)

Un **espacio de nombres** es el conjunto de relaciones entre nombres (identificadores) y los objetos a los que hacen referencia, definido dentro de un contexto.

En Python, cada **módulo** crea su propio espacio de nombres, lo que permite que distintos módulos tengan funciones o variables, potencialmente con el mismo nombre, de manera aislada.

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 (namespaces)

Al **importar** algo, lo incorporamos al espacio de nombres donde se ejecuta nuestro código.

Si añadimos más de un objeto con el mismo identificador, el último en ser importado o definido **sobrescribirá** al anterior.

```
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:

Por lo tanto, un espacio de nombres solo puede contener **un objeto por identificador.**

Para evitar conflictos, podemos usar **alias** (o seudónimos) al importar, reemplazando el identificador con la palabra "**as**". Por ejemplo:

```
# 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
```

Biblioteca estándar

Un módulo en Python puede importar otros módulos de manera sencilla, ya sea que se encuentren en la misma ubicación en el disco o que formen parte de la **biblioteca estándar** incluida con Python.

Ejemplos de módulos estándar:

datetime → permite trabajar con fechas y horas.

hashlib → funciones criptográficas.

itertools → herramientas para crear y manipular iteradores de forma eficiente.

os → interactuar con el sistema operativo (directorios, variables de entorno, etc.).

Python Module Index — Python 3.12 documentation

Continuando

En Python, un módulo puede contener dos tipos principales de contenido:

Definiciones: funciones, clases o variables que otros módulos pueden reutilizar.

Código ejecutable: instrucciones que se ejecutan inmediatamente cuando el módulo se carga

Esto significa que, al **importar** un módulo con import, no solo se ponen a disposición sus funciones y variables, sino que también se ejecuta **todo el código** que no esté dentro de una función o clase.

Importando

Como vimos en la clase anterior, Python solo necesita que el código exista en el momento en que se ejecuta. Podemos **importar** módulos o funciones en cualquier parte del archivo.

Sin embargo, es una buena práctica colocar **todas las dependencias e importaciones** al inicio del módulo. Esto facilita la lectura y evita confusión al mantener un orden coherente.

```
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

Es posible importar todos los atributos de un módulo a la vez, pero no es recomendado:

- No queda claro de dónde provienen las funciones o variables.
- Puede incluir elementos no pensados como públicos.
- Puede sobrescribir nombres existentes en el espacio de nombres.

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

¡siempre sé explícito!

Recomendacion

Para evitar confusiones y problemas con módulos y dependencias, es recomendable seguir patrones establecidos en guías de estilo como **PEP 8** o la **Google Python Style Guide.**

Estas guías ofrecen convenciones claras sobre cómo organizar importaciones, nombrar módulos y mantener un código legible y coherente.

styleguide | Style guides for Google-originated open-source projects

Módulos como scripts

En Python, como en cualquier lenguaje, un **script** es un archivo que se ejecuta de principio a fin con un objetivo específico.

Un **módulo** en Python, por defecto, puede cumplir este rol: por un lado, exponer funciones y recursos reutilizables, y por otro, proporcionar un flujo de ejecución cuando se ejecuta directamente con el intérprete.

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 declarado en un módulo tanto cuando **se lo ejecuta como script**, como 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

¿Podemos controlar qué partes de un módulo se ejecutan según la forma en que se utilice?

- ¿Es posible ejecutar cierto código **solo** cuando el módulo se usa como **script**?
- ¿O ejecutar cierto código **solo** cuando se importa desde otro módulo?

La respuesta es **sí**.

Para ello, Python nos proporciona la variable especial __name__, que nos permite diferenciar si el archivo se está ejecutando directamente o si se está importando.

__name__

La variable “mágica” __name__ está disponible por defecto en cualquier módulo.

Python la define de dos maneras distintas:

- Si el módulo se ejecuta directamente, __name__ toma el valor "__main__".
- Si se importa, __name__ toma el nombre del módulo.

Gracias a esto, podemos controlar con flexibilidad qué partes del código se ejecutan según el contexto.

__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
```

Encuesta

¿Cuál será el valor que se imprimirá en la consola cuando ejecutemos **mi_programa.py**?

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

mi_programa.py

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

utilidades.py

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

Encuesta

¿Cuál será el valor que se imprimirá en la consola cuando ejecutemos **mi_programa.py**?

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

mi_programa.py

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

utilidades.py

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

__name__

Podemos usar __name__ para decidir qué código se ejecuta según si el módulo se ejecuta directamente o se importa desde otro.

```
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

Cuando un módulo importa a otro, Python lo busca siguiendo una ruta de búsqueda predeterminada.

Primero revisa la **biblioteca estándar** y luego los directorios listados en la variable del sistema **sys.path**, uno a uno:

Por defecto:

- El directorio del script principal (donde se inició el intérprete).

- Los directorios definidos en la variable de entorno PYTHONPATH.

- El directorio de instalación predeterminado de Python.

Encontrando módulos

Es importante asegurarse de que el módulo esté en una ubicación esperada.

Conocer esto es muy útil para depurar errores de importación, ya que permite verificar si Python está buscando en los lugares correctos.

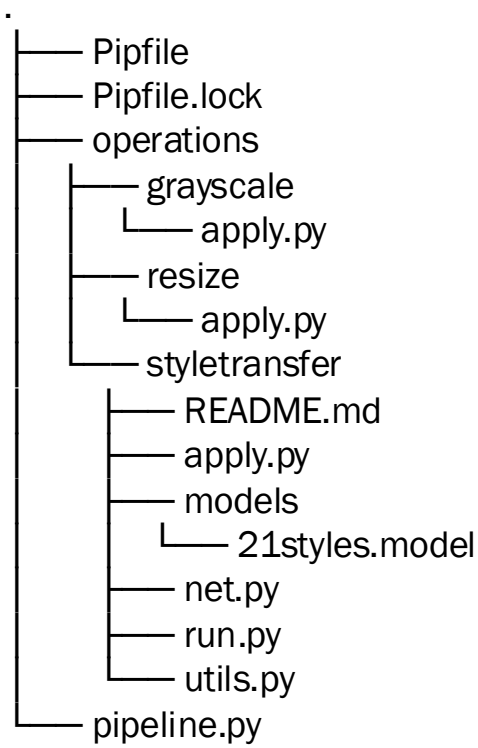
Además, podemos añadir directorios modificando **sys.path**, que no es más que una lista de Python.

Estructura de carpeta

En proyectos grandes, rara vez se trabaja con un único módulo.

Lo habitual es organizar el código en varios archivos y carpetas para facilitar su mantenimiento y permitir una colaboración más fluida.

Python brinda una forma estructurada de gestionar este tipo de proyectos mediante **paquetes.**



Paquetes

Un **paquete** es una forma de agrupar módulos de manera estructurada.

En Python, un paquete se define simplemente mediante un árbol de carpetas que organiza los módulos. Esta jerarquía se refleja en el código usando puntos para acceder a los distintos niveles.

Por ejemplo:

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

Para que Python reconozca un directorio como un **paquete**, este debe contener un archivo especial llamado __init__.py.

Este archivo puede estar vacío, pero también puede incluir código que se ejecute automáticamente cuando se importe el paquete o alguno de sus subpaquetes.

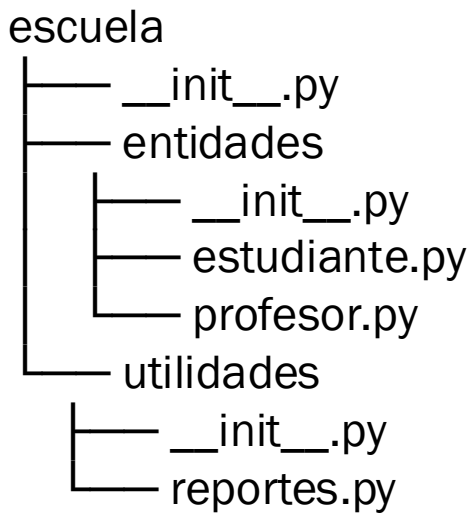
Python: Encuesta

¿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
```

Python: Encuesta

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



import escuela.entidades.estudiante

Conclusión: Módulos y paquetes

En Python, un **módulo** es un archivo que puede ser importado por otros módulos o ejecutado directamente.

Un **paquete** es una forma de agrupar módulos en una jerarquía, lo que permite organizar el código y definir un espacio de nombres estructurado.

¿Qué es una librería?

Una **librería** es cualquier módulo, paquete o conjunto de ambos que se distribuye con el objetivo de ser reutilizado en otros programas.

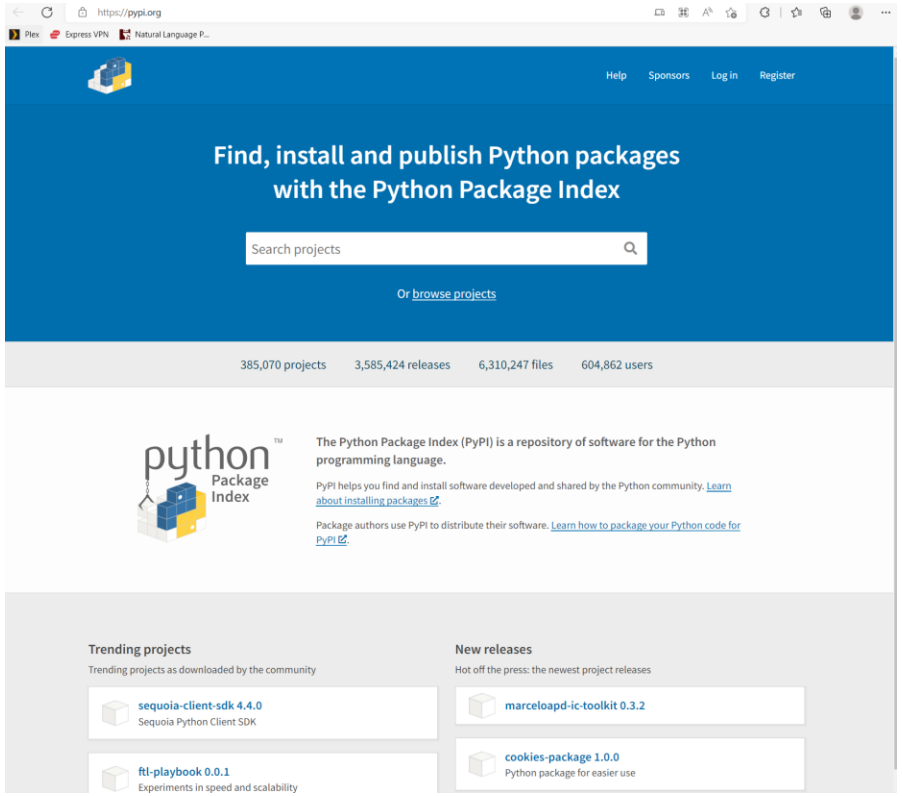
Puede ser tan simple como un solo archivo .py o tan compleja como una colección de múltiples paquetes y subpaquetes.

Librerías externas

No siempre es buena idea crear todo desde cero. En la mayoría de los casos, un proyecto necesita **dependencias externas** que ofrecen funcionalidades desarrolladas por otros.

Por ejemplo, existen librerías para analizar datos, conectarse a servidores de forma eficiente, manipular imágenes, realizar cálculos científicos, entre muchas otras cosas.

Librerías en Python



El índice público de paquetes (PyPI)

El **Python Package Index (PyPI)** es el repositorio público donde se alojan más de **300 000 paquetes** de acceso libre, listos para ser instalados y utilizados.

PyPI · The Python Package Index

Dependencias en Python: pip

En el ecosistema de Python, el instalador de librerías por defecto es **pip**, incluido en las versiones más recientes.

Para instalar una librería desde el repositorio público de Python (PyPI):

python -m pip install NombreDeLaLibreria

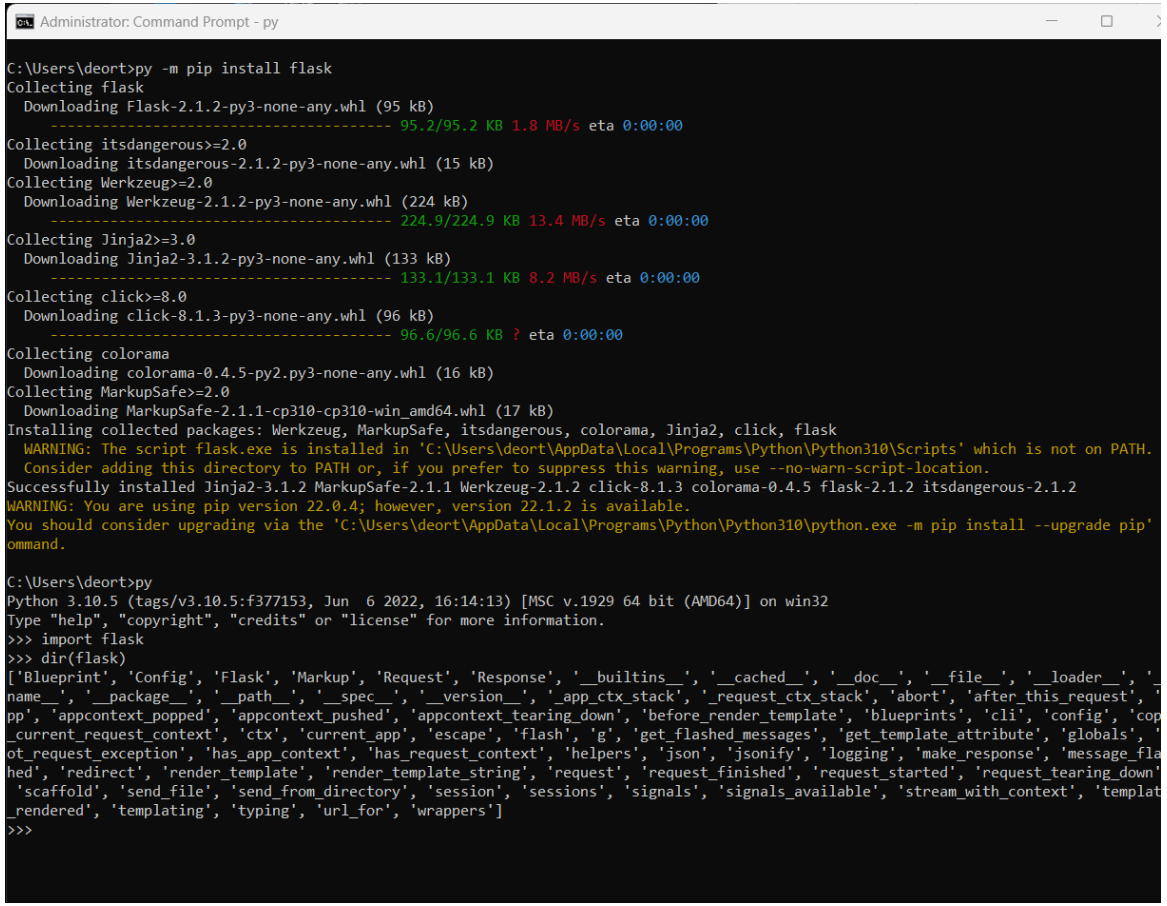
Existen muchas capacidades, como por ejemplo definir versiones exactas a instalar.

python -m pip install NombreLibreria==1.0.4

Installing Python Modules — Python 3.12 documentation

Dependencias en Python

Una vez instalada la dependencia, podemos acceder a ella con **import**, del mismo modo que lo haríamos con un modulo o paquete local.



Publicar librerías

Publicar una librería es buena idea cuando el código es reutilizable, resuelve un problema común y puede beneficiar a otros proyectos, ya sea dentro de un equipo o en la comunidad.

- Utilidades nuevas que aún no existen en la comunidad.
- Abstracciones que simplifiquen tareas para usuarios menos avanzados.
- Distribución reutilizable de componentes comunes entre proyectos de una empresa.

Publicar librerías

Publicar una librería es buena idea cuando el código es reutilizable, resuelve un problema común y puede beneficiar a otros proyectos, ya sea dentro de un equipo o en la comunidad.

- Utilidades nuevas que aún no existen en la comunidad.
- Abstracciones que simplifiquen tareas para usuarios menos avanzados.
- Distribución reutilizable de componentes comunes entre proyectos de una empresa.

Librerías: Define el objetivo

¿Qué problema resuelve tu librería?

- Interfaz mínima y estable; evita “features” innecesarias.
- Documentación: Muy importante
- Decide si será interna (repositorio Empresarial por ejemplo) o pública (PyPI).

Librerías: Estructura

Crea pyproject.toml (o setup.py)

Este archivo define metadatos (nombre, versión, autor, licencia, dependencias) y cómo construir la librería, siguiendo el estándar moderno reconocido por Python.

https://peps.python.org/pep-0518/

```
[build-system]
requires = ["setuptools", "wheel"]
build-backend = "setuptools.build_meta"

[project]
name = ”lilibreria"
version = "0.1.0"
description = "Convierte CSV a JSON"
authors = [{name=”Mi nombre"}]
```

Librerías: Entornos

tox ejecuta tests en distintas versiones de Python, asegurando que la librería funcione igual en todos los entornos soportados, de acuerdo al contrato definido en tu **pyptoject.toml**

https://tox.wiki/en/latest/

```
[tox]
envlist = py39, py310, py311

[testenv]
deps = pytest
commands = pytest
```

Librerías: Construye

Generar formatos de distribución, con **python -m build** produce los archivos que pip usará para instalar la librería en cualquier sitio.

```
python -m pip install --upgrade build
python -m build
# dist/milib-0.1.0.tar.gz y dist/milib-0.1.0-py3-none-any.whl
```

Librerías: Publica

Podemos usar **twine** para publicar tanto versions experimentales, como finales.

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

Version de prueba

```
python -m twine upload --repository testpypi dist/*
```

Version final

```
python -m twine upload dist/*
```

Para publicar en PyPI necesitas una cuenta, tu código organizado con metadatos correctos, empaquetarlo con build y subirlo con twine usando un token de API.

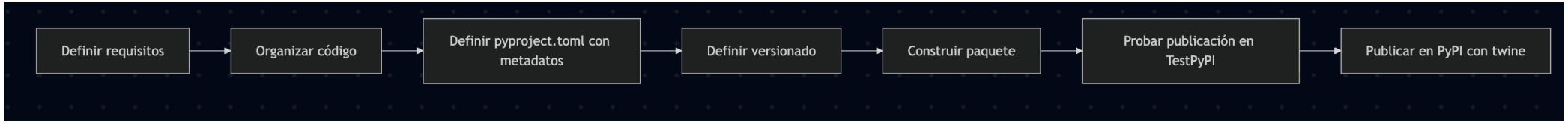
Librerías: Instala

Una vez publicada, tu librería queda disponible y las versiones anteriores se preservan en el repositorio global.

```
python –m pip install mylibreria
```

La compatibilidad y el control de versiones dependen de la configuración correcta de los metadatos.

Librerías



En entornos reales, tanto para librerías como para proyectos generales, el flujo incluye un ciclo **CI/CD** que permite promover versiones de forma controlada y eficiente, con automatización de pruebas, linters y compilación antes de publicar. Veremos este proceso en detalle más adelante.

ANOTACIONES DE TIPADO EN PROYECTOS DE PYTHON

Recordando tipos

Python es un lenguaje **dinámico**, lo que significa que las variables pueden contener valores de cualquier tipo y ese tipo puede cambiar en cualquier momento durante la ejecución del script.

```
a = 1

a = “abcd”
```

Anotaciones de tipo

Aunque Python no verifica el tipo de las variables en tiempo de ejecución, es útil añadir **anotaciones de tipo.**

Permite que los desarrolladores comprendan mejor el código y para que las herramientas automáticas puedan detectar errores que podrían pasar desapercibidos.

Introducidas en Python 3.5

https://peps.python.org/pep-0484/

Anotaciones de tipo

En lenguajes como **C**, los tipos de datos son obligatorios y el compilador los verifica antes de generar el programa, impidiendo errores de tipo en tiempo de ejecución.

```c
#include<stdio.h>
int main()
{
   char hello[] = "hello world!";
   printf("%s\n", hello);
   return 0;
}
```

En **Python**, los tipos no son requeridos y el intérprete solo generará un error en el momento exacto en que se intente realizar una operación incompatible

Anotaciones de tipo

La ausencia de tipos obligatorios en Python simplifica el código y, en algunos casos, la complejidad.

Sin embargo, las anotaciones de tipo **aportan claridad y eliminan ambigüedades**, ayudando a prevenir errores como que una variable destinada a ser un entero termine almacenando una cadena, o que se pasen argumentos incorrectos a una función.

Beneficios más evidentes a medida que aumenta la complejidad de un proyecto.

Anotaciones de tipo

¿Entonces para qué usarlas?

- Mejoran la legibilidad y claridad del código.
- Entorno de desarrollo (VS Code, PyCharm, etc.) con verificación y autocompletado más precisos.
- Documentación declarativa integrada en el código.
- Habilitan el uso de herramientas para realizar chequeos estáticos automáticos.

Anotaciones de tipo

Desde Python 3.5, podemos definir tipado en las propiedades de nuestro código de la siguiente manera:

```python
cadena_de_texto: str = "hello world!"

def sumar(x: int, y: int) -> int:
    return x + y

valor_sumado: int = sumar(7, 4)
```

En Python, las anotaciones de tipo se indican con **dos puntos (:)** para definir el tipo de una variable o parámetro, y con **una flecha (->)** para especificar el tipo de valor que retorna una función.

Anotaciones de tipo: Alerta

Usar anotaciones de tipo permite que el IDE, el linter o el entorno de desarrollo detecten y señalen problemas como alertas tempranas, antes de que el código llegue a los usuarios.

Más adelante veremos cómo automatizar estos chequeos dentro del flujo de desarrollo.

Combinando tipos

Además de los tipos básicos como int, str, float o dict, también es posible combinarlos para definir tipos más complejos. Por ejemplo, podemos indicar que una variable acepte **varios tipos** usando anotaciones como Union o el operador |.

```
int_o_cadena: str | int = 40
```

Tipado en clases

```python
class Empleado:
    def __init__(self, nombre, edad, salario):
        self.nombre = nombre
        self.edad = edad
        self.salario = salario
        self.proyectos = []

    def asignar_proyecto(self, proyecto):
        self.proyectos.append(proyecto)

    def obtener_detalles(self):
        detalles = f"Nombre: {self.nombre}, Edad: {self.edad},
Salario: {self.salario}, Proyectos: {',
'.join(self.proyectos)}"
        return detalles
```

```python
from typing import List

class Empleado:
    def __init__(self, nombre: str, edad: int, salario:
float):
        self.nombre: str = nombre
        self.edad: int = edad
        self.salario: float = salario
        self.proyectos: List[str] = []

    def asignar_proyecto(self, proyecto: str) -> None:
        self.proyectos.append(proyecto)

    def obtener_detalles(self) -> str:
        detalles = f"Nombre: {self.nombre}, Edad: {self.edad},
Salario: {self.salario}, Proyectos: {',
'.join(self.proyectos)}"
        return detalles
```

Encuesta

¿Resultado de ejecutar el siguiente programa?

mi_programa.py

```
mi_variable: int = 50
print(mi_variable)
mi_variable = "daniel"
print(mi_variable)
```

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

Encuesta

¿Resultado de ejecutar el siguiente programa?

mi_programa.py

```
mi_variable: int = 50
print(mi_variable)
mi_variable = "daniel"
print(mi_variable)
```

```
doc:~$ python3 mi_programa.py
50
daniel
```

A profundidad

En Python, el tipado ofrece un mundo más completo: podemos crear **uniones**, definir **tipos dinámicos**, usar **tipos condicionales** y muchas otras combinaciones para describir con precisión cómo debe comportarse nuestro código.

https://docs.python.org/3/library/typing.html

Laboratorio

El Github incluye el segundo laboratorio para repasar los conceptos.

https://github.com/danoc93/ista-python-analisis-2025

Aviso:

Los laboratorios son opcionales, pero cada uno que se complete y suba al repositorio personal del curso otorgará un **1,5% extra** en la nota final.