Diseño de aplicaciones

Anotaciones de tipo.

Las anotaciones de tipo, también conocidas como **type hints**, son una característica de Python que se introdujo a partir de la versión 3.5. Se utilizan para proporcionar información adicional sobre los tipos de datos de las variables, argumentos de funciones y valores de retorno.

> Esta característica lleva muchos años desarrollándose entre versiones de Python y ha evolucionado rápidamente en las últimas versiones. Estos apuntes corresponden a la versión de Python 3.11 y es posible que algunas características no estén disponibles en versiones anteriores.

La sintaxis básica para proporcionar anotaciones de tipo es colocar el nombre de la variable, argumento o valor de retorno, seguido de dos puntos y luego el tipo de datos esperado. Por ejemplo, si quisieras anotar una variable llamada `edad` como un entero, podrías hacerlo de la siguiente manera:

```
edad: int = 25
```

Aquí, `int` es la anotación de tipo para el tipo de datos entero. Se puede utilizar con cualquier tipo de dato, incluidas clases definidas.

```
compra: list = ['huevos', 'patatas', 'cebollas']
codigos_postales: dict = {30202: 'Cartagena', 30203: 'Cartagena', 30003:
'Murcia'}
auto : Coche = Coche('BMW', '44522HAF')  # Tipo inventado
```

Las anotaciones de tipo se pueden utilizar en funciones de la misma manera que se utilizan para variables. La sintaxis es similar a la de una variable, pero se coloca entre los paréntesis de la definición de la función. Por ejemplo, si quisieras definir una función llamada `doble` que toma un entero y devuelve el `doble` de ese entero, podrías hacerlo de la siguiente manera:

```
def doble(numero: int) -> int:
    "Devuelve el doble de un número."
    return numero * 2
```

Aquí, `numero: int` es la anotación de tipo para el argumento `numero`, y `-> int` es la anotación de tipo para el valor de retorno de la función.

> Cuando queramos especificar que las funciones no devuelven nada podemos indicarlo con `-> None`.

Las anotaciones de tipo son opcionales en Python y no afectan el funcionamiento de tu código. Sin embargo, son una herramienta útil para ayudar a documentar tu código y hacerlo más fácil de entender y mantener. Además, algunas herramientas de análisis de código, como `mypy`, pueden **usar anotaciones de tipo para detectar errores potenciales en tu código antes de que se ejecute**. Los entornos de desarrollo, como vscode, cuentan extensiones que te brindan autocompletado y sugerencias en función del tipo de la variable que utilizas.

> **No es necesario indicar los tipos en cada variable utilizada**. Si haces `x = 10`, se sobrentiende que x es un número entero.

Tipos compuestos.

A veces tendremos la necesidad de usar variables y argumentos cuyo tipo que pueden contener otros. Por ejemplo, una lista de enteros o un diccionario de *strings*. Para especificar la composición de un tipo utilizamos los corchetes `[]`. Veamos algunos ejemplos.

Listas:

```
def mitad_rango(numeros: list[float]) -> float
    "Devuelve el punto medio del rango en una lista de números".
    return (max(numeros)+min(numeros))/2
```

Indicamos que la lista `numeros` es una lista de números en coma flotante.

Diccionarios:

```
notas : dict[str, int] = {}
notas['Amanda'] = 8
notas['Elías'] = 9
notas['Mario'] = 5
```

Indicamos que `notas` es un diccionario que usa como claves `str` y como valores `int`.

Tuplas:

```
def maxmin(valores: list[int]) -> tuple[int, int]:
    "Devuelve el máximo y el mímimo de una lista"
    return max(valores), min(valores)
```

Con las tuplas indicamos el número y tipo de elementos. En esta caso indicamos que la función devuelve dos números enteros.

Unión de tipos.

Cuando queremos indicar que una variable, argumento, etc. puede ser de más de un tipo utilizamos el operador de unión: `|`.

```
def convertir(dato: str|int) -> float|None:
    "Convierte un número o texto a coma flotante."
    try:
        return float(dato)
    except (ValueError, TypeError):  # Agrupación de errores.
        return None
```

En este ejemplo indicamos que la función acepta *strings* o entenos y que devuelve o bien números en coma flotante o `None`.

Alias de tipo.

Se pueden definir alias para tipos. Son útiles para simplificar firmas de tipo complejas.

```
```

```python
ColorRGB = tuple[int,int,int]
Imagen = list[list[ColorRGB]]

def detectar_color(imagen: Imagen, color: ColorRGB) -> bool:
    "Indica si aparece el color en la imagen"
    for fila in imagen:
        for pixel in fila:
            if color == pixel:
                return True
    return False
```

Módulo `typing`.

El módulo `typing` ofrece funcionalidades avanzadas para anotaciones de tipo. Incluye varias clases y funciones que se pueden usar para anotar tipos de datos más complejos y para crear nuestros propios tipos.

A continuación, se verán algunos de los miembros más usados de `typing`.

Any

Con `Any` se indica que se acepta cualquier tipo.

```python
from typing import Any
import random

def imprimir_bonito(obj: Any) -> None:
    "Como print pero bonito."
    decoracion = random.choices("💐🌼🌻✨💫🦄💖❤️💝💘🌷🌹🌸🌺", k=2)
    print(decoracion[0], obj, decoracion[1])
```

Self

Para devolver objetos instancias de la clase donde se utiliza.

```python
from typing import Self
import json

class Contacto:
    "Contiene los datos de contacto de alguien."

    def __init__(self, nombre: str, apellido: str, edad: str, email: str):
        self.nombre = nombre
        self.apellido = apellido
        self.edad = edad
        self.email = email
```

```
    @classmethod
    def from_json(cls, archivo: str) -> Self:
        "Carga un contacto desde un json."
        with open(archivo, 'r') as f:
            datos = json.load(f)
        return cls(**datos)
```

En este caso `from_json` ha devuelto una instancia de `Contacto` y al anotar el retorno como `Self` damos información de que lo que devuelve el método es efectivamente una instancia de `Contacto`.

`Callable`

La clase `Callable` se utiliza para anotar funciones que toman ciertos argumentos y devuelven un valor. Puedes utilizar esta clase para especificar los tipos de argumentos y valores de retorno que la función espera.

```
from typing import Callable

def suma(a: int, b: int) -> int:
    "Devuelve la suma de dos números."
    return a + b

def operar(a: int, b: int, f: Callable[[int, int], int]) -> int:
    "Aplica una función a dos elementos."
    return f(a, b)

resultado = operar(2, 3, suma)
```

En este ejemplo, la función `suma` toma dos argumentos enteros y devuelve un entero. La función `operar` también toma dos argumentos enteros y una función que toma dos argumentos enteros y devuelve un entero.

Variables de tipo.

Las variables de tipo `TypeVar` sirven tanto como de parámetros para tipos genéricos como para definición de funciones genéricas.

```
T = TypeVar('T')  # Puede ser cualquier cosa.
S = TypeVar('S', bound=str)  # Puede ser cualquier tipo de str.
A = TypeVar('A', str, bytes)  # Debe ser str o bytes.

def repetir(x: T, n: int) -> list[T]:
    """Devuelve una lista de n elementos iguales."""
    return [x]*n
```

También existe una variable de especificación de parámetros `ParamSpec` que es una versión especializada de `TypeVar`.

`Protocol`.

La clase `Protocol` se utiliza para anotar objetos que cumplen con un conjunto específico de métodos o atributos. Puedes utilizar esta clase para especificar qué métodos o atributos debe tener un objeto para cumplir con una interfaz determinada.

```
from typing import Protocol

class Persona(Protocol):
    "Representa los atributos que debe tener una persona."
    nombre: str
    apellido: str

def saludar(persona: Persona) -> str:
    "Saluda a una persona."
    return f"Hola, {persona.nombre} {persona.apellido}!"

class Empleado:
    "Representa a un empleado."
    def __init__(self, nombre: str, apellido: str, salario: float):
        self.nombre = nombre
        self.apellido = apellido
        self.salario = salario

print(saludar(Empleado("Juan", "Pérez", 2000)))
```

Tipado con clases abstractas.

Se pueden usar clases abstractas para anotar tipos de variables que cumplan con una serie de requisitos en forma de métodos y atributos. En muchos casos ahorra tener que definir nuestros propios tipos con `typing.Protocol`.

En la *Python Standard Library* existe un módulo `collections.abc` donde hay definidas clases abstractas que pueden usarse para probar si una clase proporciona una interfaz específica. Veamos algunos ejemplos recurrentes.

- `Iterable`: se utiliza para anotar objetos que se pueden recorrer con un bucle for. Puedes utilizar esta clase para especificar el tipo de elementos que se pueden recorrer en el objeto.

  ```
  from collections.abc import Iterable

  def media(lista: Iterable[int|float]) -> float:
      "Devuelve la media de un objeto iterable."
      resultado = 0
      elementos = 0
      for elemento in lista:
          resultado += elemento
          elementos += 1
      return resultado/elementos
  ```

```
print('Con una lista', media([1, 2, 3]))
print('Con una tupla', media((1, 2, 3)))
print('Con un range', media(range(1,4)))
```

- `Iterator`: se utiliza para anotar objetos que generan una secuencia de elementos y permiten iterar sobre ellos. Puedes utilizar esta clase para especificar el tipo de elementos que se generan y el valor devuelto por el método `__next__`.

```
from collections.abc import Iterator

class ContadorBinario:
    """Devuelve un rango con la
    representación de los números en binario."""

    def __init__(self, inicio: int, fin: int):
        self.inicio = inicio
        self.fin = fin

    def __iter__(self) -> Iterator[str]:
        self.actual = self.inicio
        return self

    def __next__(self) -> str:
        if self.actual >= self.fin:
            raise StopIteration
        resultado = self.actual
        self.actual += 1
        return resultado

contador = Contador(0, 3)
for i in contador:
    print(i)
```

- `Sequence`: se utiliza para anotar objetos que se comportan como secuencias de elementos, es decir, que se pueden acceder por índice y se pueden recorrer. Puedes utilizar esta clase para especificar el tipo de elementos que se pueden acceder por índice.

```
from collections.abc import Sequence

def mediana(secuencia: Sequence[int]) -> int:
    "Devuelve la mediana de una distribución."
    ordenada = sorted(secuencia)
    return ordenada[len(ordenada)//2]

print(mediana([1, 2, 3, 4, 5]))  # -> Ok
print(mediana({1:'a', 2:'b'}))  # -> Nope
```

- `Mapping`: se utiliza para anotar objetos que se comportan como diccionarios, es decir, que tienen una clave y un valor asociado. Puedes utilizar esta clase para especificar el tipo de las claves y de los valores en un diccionario.

```
from collections.abc import Mapping

def capitalizar(diccionario: Mapping[str, str]) -> Mapping[str, str]:
    return {clave: valor.capitalize() for clave, valor in
diccionario.items()}

print(capitalizar({"uno": "primero", "dos": "segundo", "tres":
"tercero"}))
```

Puedes ver la colección completa en la documentación oficial de `collections.abc`.

Ejercicio de clase.

En la carpeta `scr` encontrarás el programa de logística de la sesión 4. Introduce las anotaciones de tipo que has aprendido.

Dataclasses.

Las ***dataclasses*** son una funcionalidad introducida en Python 3.7 que permiten crear clases para almacenar datos de manera sencilla **utilizando las funcionalidades de anotaciones de tipado**. Una *dataclass* es esencialmente una clase que tiene menos código repetitivo y que automáticamente proporciona una serie de métodos especiales, como `__init__()`, `__repr__()`, `__eq__()`, entre otros.

Para crear una *dataclass*, se utiliza el decorador `@dataclass` y se define la clase con las variables que se desean almacenar. Por ejemplo:

```
from dataclasses import dataclass

@dataclass
class Persona:
    nombre: str
    edad: int
    altura: float

manuel = Persona('Manuel', 23, 1.77)
```

En este ejemplo, se ha creado una clase `Persona` que tiene tres variables: `nombre`, `edad` y `altura`. La clase está decorada con `@dataclass`, lo que indica que es una Dataclass.

Con esto, automáticamente se generarán los métodos especiales `__init__()`, `__repr__()`, `__eq__()`, entre otros, que permiten instanciar objetos de la clase, representar la información de la clase de manera legible y comparar objetos de la clase, respectivamente.

Además de las variables que se definen en una *dataclass*, también es posible utilizar la función `field` para personalizar aún más el comportamiento de los atributos en la clase.

Por ejemplo, con la función `field` se puede establecer un valor por defecto para una variable, como se muestra en el siguiente ejemplo:

```
from dataclasses import dataclass, field

@dataclass
class Coche:
    "Contiene los datos de un coche."
    marca: str
    modelo: str
    año: int
    precio: float = field(default=20000.0)
    color: str = field(default='negro')

coche = Coche(marca='Toyota', modelo='Corolla', año=2020)
print(coche)
```

En este caso, las variables `precio` y `color` tienen valores por defecto que se utilizarán si no se proporcionan esos valores al instanciar un objeto de la clase.

Además de `default`, existen otros argumentos que se pueden utilizar con la función field, como `init`, que indica si la variable debe ser incluida en el método `__init__()`, o `repr`, que indica si la variable debe ser incluida en el método `__repr__()`.

Las *dataclass* también incluyen métodos especiales como `__post_init__()`, que se ejecuta después de crear la clase y que permite por ejemplo calcular alguna variable a partir de los valores de los atributos sí inicializados.

Veamos un ejemplo más completo:

```
from dataclasses import dataclass, field

@dataclass
class Animal:
    nombre: str
    edad: int
    especie: str
    color: str = field(default='blanco', repr=False)
    sonido: str = field(init=False)

    def __post_init__(self):
        if self.especie == 'perro':
            self.sonido = 'guau'
        elif self.especie == 'gato':
            self.sonido = 'miau'
        else:
            self.sonido = 'desconocido'
```

```
    def hacer_sonido(self):
        print(self.sonido)

perro = Animal(nombre='Fido', edad=3, especie='perro', color='marrón')
gato = Animal(nombre='Michi', edad=2, especie='gato')
dino = Animal(nombre='Dino', edad=10, especie='dinosaurio', color='verde')

print(perro)
gato.hacer_sonido()
```

En este ejemplo, estamos creando una clase llamada `Animal` que tiene cinco atributos: `nombre`, `edad`, `especie`, `color` y `sonido`.

El atributo `color` tiene un valor por defecto y la opción `repr=False` para evitar que se muestre en el método `__repr__`.

El atributo `sonido` tiene el decorador `init=False` para indicar que no debe ser inicializado en el constructor `__init__`.

En el método `__post_init__`, que se llama después de la inicialización de los atributos, comprobamos la especie del animal y establecemos el valor del atributo sonido en consecuencia.

La clase también tiene un método `hacer_sonido()` que imprime el sonido del animal, pues podemos incluir cualquier método o propiedad sin afectar a la *dataclass*.

Existen más funcionalidades asociadas a las *dataclasses* como clases inmutables (*frozen*) y más métodos de construcción. En la documentación oficial encontrarás todo la información.

> Documentación sobre *dataclasses*: https://docs.python.org/3/library/dataclasses.html

Ejercicio de clase.

En la carpeta `scr` encontrarás el programa de logística de la sesión 4. Convierte la clase `Pallet` y `Pedido` en *dataclasses*.

Interfaz de línea de comandos.

Una **interfaz de línea de comandos** (**CLI**, por sus siglas en inglés) es una forma de interactuar con una aplicación o programa a través de comandos escritos en la línea de comandos del sistema operativo. Las CLI son comunes en sistemas operativos Unix y Linux, aunque también se utilizan en Windows.

En Python, una interfaz de línea de comandos se puede implementar utilizando el módulo `argparse`. Este módulo permite crear un analizador de argumentos que puede interpretar los argumentos pasados a través de la línea de comandos y ejecutar las acciones correspondientes.

> Una CLI puede darse durante la ejecución del programa pero el módulo `argparse` está especializado en pasar argumentos desde la terminal al comenzar la ejecución (se los pasa realmente el sistema operativo).

El módulo `argparse` proporciona varias funciones y clases que permiten la definición de los argumentos que la CLI soporta, y cómo estos argumentos deben ser procesados.

Funcionalidad principal.

> Lo que se muestra a continuación está extraido de la documentación oficial.

El soporte del módulo `argparse` para las interfaces de líneas de comandos es construido alrededor de una instancia de `ArgumentParser`. Este es un contenedor para las especificaciones de los argumentos y tiene opciones que aplican al analizador en su conjunto:

```
parser = argparse.ArgumentParser(
                    prog = 'Nombre del programa',
                    description = 'Explicación general',
                    epilog = 'Línea final de la ayuda.')
```

El método `ArgumentParser.add_argument()` añade argumentos individuales al analizador. Soporta argumentos posicionales, opciones que aceptan valores, y *flags* de activación y desactivación:

```
parser.add_argument('archivo')           # argumento posicional
parser.add_argument('-c', '--count')     # argumento opcional que acepta un
valor
parser.add_argument('-v', '--verbose',
                    action='store_true')  # argumento 'flag', solo
comprueba si está o no está (True/False).
```

El método `ArgumentParser.parse_args()` ejecuta el analizador y coloca los datos extraídos en un objeto `argparse.Namespace`:

```
args = parser.parse_args()
print(args.archivo, args.count, args.verbose)  # Accedes a los argumentos
como si fueran atributos.
```

Si guardamos el código de arriba en un módulo, lo podremos ejecutar desde la terminal (ejemplo para linux, `$` representa la entrada de usuario):

```
$ python3 ejemplo.py -c 10 -v archivo.txt
archivo.txt 10 True
$ python3 ejemplo.py --help
usage: Nombre del programa [-h] [-c COUNT] [-v] archivo

Explicación general

positional arguments:
  archivo
```

```
options:
  -h, --help            show this help message and exit
  -c COUNT, --count COUNT
  -v, --verbose

Línea final de la ayuda.
$ python3 ejemplo.py
usage: Nombre del programa [-h] [-c COUNT] [-v] archivo
Nombre del programa: error: the following arguments are required: archivo
$ python3 ejemplo.py archivo.txt
archivo.txt None False
```

> Automáticamente se añade un argumento de ayuda `-h` (o `--help` en su forma larga).

Veamos un ejemplo más funcional. El siguiente programa lee un archivo `csv` dado por línea de comandos y devuelve los estadísticos de una columna.

```
import pandas as pd
import argparse


def extraer_stats(datos: pd.Series, percentiles: list[float], indice: bool
= False) -> dict[str, float | tuple]:
    "Extrae estadísticas de una serie de datos de pandas."
    stats = {}
    stats["media"] = round(datos.mean(), 2)
    stats["sd"] = round(datos.std(), 2)
    if indice:
        stats["min"] = datos.min(), datos.idxmin()
        stats["max"] = datos.max(), datos.idxmax()
    else:
        stats["min"] = (datos.min(),)
        stats["max"] = (datos.max(),)
    stats["mediana"] = datos.median()
    for percentil in percentiles:
        stats[f"p{percentil:.2f}"] = datos.quantile(percentil)
    return stats


def imprimir_stats(datos_stats: dict[str, float | tuple], columna: str) ->
None:
    "Imprime las estadísticas."
    print(f"----------------------------------------")
    print(f'Columna "{columna}"')
    print(f"----------------------------------------")
    print("Media", datos_stats["media"], sep="\t\t")
    print("SD", datos_stats["sd"], sep="\t\t")
    print("Mínimo", *datos_stats["min"], sep="\t\t")
    print("Máximo", *datos_stats["max"], sep="\t\t")
    print("Mediana", datos_stats["mediana"], sep="\t\t")
```

```
    print("-- Percentiles --")
    for clave in filter(lambda clave: clave.startswith("p"), datos_stats):
        print(clave[1:], datos_stats[clave], sep="\t\t")


if __name__ == "__main__":
    parser = argparse.ArgumentParser(
        prog="Estadística",
        description="Extrae datos estadísticos de una columna de un archivo
csv.",
        epilog="Si no existe encabezado y todos los datos de la columna no
son números el programa fallará.",
    )
    parser.add_argument("archivo_csv", help="Nombre del archivo .csv")
    parser.add_argument("columna", help="Nombre de la columna.")
    parser.add_argument("-p", "--percentiles", help="Calcula los
percentiles.", default="0.25,0.5,0.75")
    parser.add_argument(
        "-i", "--index", help="Te devuelve el índice de los elementos del
máximo y mínimo.", action="store_true"
    )

    args = parser.parse_args()
    percentiles: list[float] = [*map(float, args.percentiles.split(","))]

    datos = pd.read_csv(args.archivo_csv)
    stats = extraer_stats(datos[args.columna], percentiles, args.index)
    imprimir_stats(stats, args.columna)
```

> Pruébalo con el dataset `notas_alumnos`.

El módulo `argparse` tiene muchas funcionalidades y es imposible verlas todas en un curso intensivo. Recurre a la documentación oficial para saber más y hacer interfaces de línea de comandos más complejas.

Paquetes.

Los paquetes en Python son una forma de organizar y estructurar los módulos relacionados en una jerarquía de carpetas. Un paquete es una carpeta que contiene uno o varios módulos y, opcionalmente, otros subpaquetes. Los paquetes se utilizan para organizar grandes proyectos de Python en un conjunto coherente de módulos y subpaquetes.

Los paquetes se crean mediante la inclusión de un archivo especial llamado `__init__.py` en la carpeta que contiene los módulos del paquete. Este archivo puede estar vacío o puede contener código Python, y se ejecuta cuando se importa el paquete. Un ejemplo de estructura de carpeta de un paquete es:

```
paquete/
    __init__.py
    main.py
    modulo.py
    utils/
        __init__.py
```

```
        file.py
        string.py
    management/
        __init__.py
        users.py
        auth.py
    tests/
        __init__.py
        test_file.py
        test_string.py
```

En este ejemplo, el paquete principal se llama `paquete`, que contiene tres subpaquetes: `utils`, `management` y `tests`. El archivo `__init__.py` en el directorio raíz del paquete `paquete` es necesario para indicar que es un paquete de Python.

El archivo `main.py` se encuentra en el directorio raíz del paquete `paquete` y podría contener el punto de entrada para el programa. Algunas funciones y clases podrían estar en `modulo`. Los módulos de utilidad, como `file.py` y `string.py`, se encuentran en el subpaquete `utils`; los módulos relacionados con la administración, como `users` o `auth`, se encuentran en el subpaquete `management` y los archivos de prueba, como `test_file.py` y `test_string.py`, se encuentran en el subpaquete tests.

> No vamos a trabajar con tests en este curso por falta de tiempo, pero es una skill muy importante para cualquier tipo de programador. Se recomienda echar un vistazo a la documentación del módulo estándar para testing `unittest`.

Para importar un módulo desde un paquete, se utiliza la siguiente sintaxis:

```
import paquete.modulo
```

También se puede utilizar la siguiente sintaxis para importar un objeto específico de un módulo dentro de un paquete:

```
from paquete.modulo import objeto
```

Si el paquete contiene subpaquetes, se puede acceder a los módulos dentro de los subpaquetes utilizando la siguiente sintaxis:

```
import paquete.subpaquete.modulo
import paquete.utils.file  # En nuestro ejemplo.
```

También es posible dar un alias un paquete o un objeto específico utilizando la siguiente sintaxis:

```
import paquete as alias
from paquete.modulo import objeto as alias
```

Referencias internas en paquetes.

Cuando se estructuran los paquetes en subpaquetes (como en el ejemplo `utils`), puedes usar import absolutos para referirte a `submódulos` de paquetes hermanos. Por ejemplo, si el módulo `paquete.management.users` necesita usar el módulo `string` en el paquete `paquete.utils`, puede hacer:

```
from paquete.utils import string
```

También puedes escribir *imports* relativos. Estos *imports* usan puntos para indicar los paquetes actuales o paquetes padres involucrados en el *import* relativo. Si por ejemplo estuvieramos trabajando en el módulo `management.users`, podíamos hacer:

```
from . import auth
from .. import modulo
from ..utils import string
```

Nótese que los *imports* relativos se basan en el nombre del módulo actual. Ya que el nombre del módulo principal es siempre "**main**", los módulos pensados para usarse como módulo principal de una aplicación Python siempre deberían usar *import* absolutos.

`__init__.py`

Como ya se ha mencionado, el archivo `__init__.py` es un archivo especial en Python que se utiliza para indicar que un directorio debe ser tratado como un paquete de Python.

Cuando Python importa un paquete, busca un archivo `__init__.py` en el directorio del paquete y lo ejecuta. El archivo `__init__.py` puede contener código y se utiliza para inicializar el paquete y realizar tareas de configuración, como la importación de módulos, la definición de variables y funciones compartidas, y la configuración de los parámetros del paquete.

Algunas de las cosas que se suelen hacer en el archivo `__init__.py` son:

- **Importar submódulos**: se puede importar módulos y subpaquetes dentro del paquete para que estén disponibles para su uso en otros módulos.
- **Definir variables globales**: se pueden definir variables globales en el archivo `__init__.py` para que estén disponibles en todo el paquete.
- **Configurar parámetros**: se pueden establecer variables de configuración y parámetros en el archivo `__init__.py` para que estén disponibles en todos los módulos del paquete.
- **Ejecutar código de inicialización**: se puede ejecutar código de inicialización en el archivo `__init__.py`, como la conexión a bases de datos o la configuración de bibliotecas externas.

- **Controlar la API del paquete**: se puede controlar la API del paquete en el archivo `__init__.py`, determinando qué módulos y funciones están disponibles para los usuarios que importen el paquete.

Ejecutar módulos/paquetes.

Si se desea ejecutar un módulo dentro de un paquete, se puede hacer a través de la siguiente sintaxis dentro de la línea de comandos.

```
py -m paquete.modulo  # En Windows
python3 -m paquete.modulo  # En Linux/Mac
```

Si, además, dentro del paquete llamamos a un módulo `__main__.py`, este se convertirá en el punto de entrada del programa. En cuyo caso podríamos hacer:

```
py -m paquete  # En Windows
python3 -m paquete  # En Linux/Mac
```

Realizando este comando en la terminal se ejecutaría el código de `__main__.py`.

Ejercicio de clase.

En la carpeta `scr` encontrarás el programa de logística de la sesión 4. Haz las siguientes modificaciones:

- Crea un módulo `io.py` e introduce la función `cargar_pedidos(...)` del módulo `main.py` y añade la función `redactar_informe(...)` que genere un documento de texto con un informe de los camiones desglosados por destino.
- Dale estructura de paquete (`gestorlogis`) con `__init__.py` y añade un `__main__.py` (puede sustituir a `main.py` si quieres). Modificalo para que llame también a `redactar_informe`.
- Añade una interfaz de línea de comandos con `argparse` donde le pases los siguientes parámetros.
    - Origen del archivo de pedidos. (Obligatorio)
    - Destino del archivo del informe. (Obligatiorio)
    - *Verbose*, es decir, que muestre por pantalla la información del informe. (Opcional, con `-v, --verbose`).

Debes poder ejecutar el programa en la línea de comandos del siguiente modo:

```
python3 -m gestorlogis pedidos.json informe.txt  # En Linux
```

```
py -m gestorlogis pedidos.json informe.txt  #  En Windows
```

pip

`pip` es el gestor de paquetes por defecto para Python que permite instalar, actualizar y desinstalar paquetes o módulos de Python, así como manejar sus dependencias. `pip` también permite buscar y listar información sobre paquetes disponibles en el índice de paquetes de Python (PyPI).

Algunas de las funcionalidades más útiles de `pip` son:

- **Instalar paquetes**: se puede instalar un paquete de Python usando `pip install nombre_paquete`. Pip automáticamente resuelve y descarga las dependencias necesarias para instalar el paquete.
- **Desinstalar paquetes**: se puede desinstalar un paquete instalado anteriormente usando `pip uninstall nombre_paquete`.
- **Actualizar paquetes**: se puede actualizar un paquete instalado a la última versión disponible usando `pip install --upgrade nombre_paquete`.
- **Listar paquetes**: se puede listar los paquetes instalados usando `pip list` o `pip freeze`.

Además, `pip` también es capaz de instalar paquetes desde archivos comprimidos o desde repositorios de versiones de control de código fuente como Git.

Entornos virtuales.

Los **entornos virtuales** en Python son herramientas que se utilizan para crear un **entorno de desarrollo aislado con sus propias dependencias y versión de Python**. Esto es útil para asegurar que los proyectos tengan un conjunto de dependencias específicas y evitar conflictos entre ellas.

Cuando se trabaja en proyectos avanzados, es común que algunos paquetes requieran de versiones concretas de otros módulos y paquetes. Si instalas todos los paquetes en el entorno global de Python es muy probable que te encuentres **paquetes con las mismas dependencias en diferentes versiones** y que al instalar uno de ellos el otro deje de funcionar.

Además, existen paquetes que solo están disponibles para versiones de Python anteriores. Por ejemplo `tensorflow` no funciona en Python 3.11 y no ofrece versiones precompiladas oficiales de la versión 3.10.

Por lo tanto, crear un entorno virtual para cada proyecto ayuda a evitar conflictos entre las diferentes dependencias y versiones de Python.

> Puedes tener varias versiones de Python en el ordenador. Para hacer eso no debes marcar 'Añadir al PATH' cuando instales en Windows porque no tu versión principal. En Linux debes compilar con el atributo `altinstall` cuando hagas `make`.

Para crear un entorno virtual en Python, podemos utilizar la herramienta venv que viene incluida en la biblioteca estándar de Python a partir de la versión 3.3. Podemos crear un nuevo entorno virtual utilizando el siguiente comando en la línea de comandos:

```
python -m venv nombre_entorno
```

Este comando creará un nuevo directorio llamado `nombre_entorno` que contiene un entorno virtual de Python.

Normalmente el nombre del entorno suele empezar por punto (`'.'`). Esto se hace para que aparezca como oculto en la navegación de carpetas por terminal.

Podemos activar este entorno virtual ejecutando el siguiente comando:

```
source myenv/bin/activate  # En Linux/Mac
```

```
myenv\Scripts\activate  # En Windows
```

Una vez activado el entorno virtual, aparecerá el nombre del entorno precediendo al *prompt* y podremos instalar dependencias específicas para nuestro proyecto usando el administrador de paquetes de Python `pip`. **Las dependencias se instalarán dentro del entorno virtual aislado, lo que significa que no afectarán a otros proyectos** o al sistema de Python en sí.

> Añadiendo `--prompt ALIAS` puedes cambiar el prefijo que aparece en la consola cuando está el entorno activado. Es especialmente útil en nombres de entorno largos o cuando llamas de forma genérica al entorno.

Cuando ejecutas un módulo o paquete con `python` estando el entorno virtual activado estarás ejecutando una copia del intérprete situada en la carpeta del entorno y todo lo que hagas, como instalar paquetes, lo harás también desde ese intérprete.

Para desactivar el entorno virtual, podemos ejecutar el siguiente comando:

```
deactivate
```

Puedes encontrar más información detallada sobre los entornos virtuales en la guía oficial.

Construye tu aplicación.

A nivel de proyecto, una aplicación de Python profesional debe cumplir con varios requisitos para garantizar su calidad, mantenibilidad y escalabilidad a largo plazo. Algunos de estos requisitos son:

- **Correcta estructura de proyecto**: el proyecto debe seguir una estructura clara y bien definida, que permita la fácil navegación y comprensión del código por parte de los desarrolladores. Idealmente, se debe seguir la estructura de paquetes clásica de Python.
- **Documentación**: el código debe estar bien documentado, con comentarios claros y concisos, y respetando las anotaciones de tipo.
- **Control de versiones**: el proyecto debe estar gestionado mediante un sistema de control de versiones, como Git, que permita el seguimiento de los cambios en el código y la colaboración entre desarrolladores.

- **Test**: el proyecto debe contar con test unitarios automatizados, que permitan verificar el correcto funcionamiento del código y detectar errores tempranamente. No lo veremos en este curso.
- **Manejar requerimientos y dependencias**: el proyecto debe incluir un archivo con los requerimientos y dependencias necesarios para su correcto funcionamiento, que permita la fácil instalación y configuración del entorno de desarrollo y producción.
- **Configuración**: el proyecto debe contar con un sistema de configuración flexible, que permita la fácil personalización del comportamiento de la aplicación en diferentes entornos y situaciones. Esto puede hacerse mediante archivos de configuración (`json`, `toml`, etc.), mediante CLI, etc.

Hay muchas manera de cumplir con estos requirimientos a la hora de construir nuestra aplicación. En **proyectos de desarrolo e innovación** la estructura de nuestro **proyecto** puede ser algo así:

```
carpeta_proyecto/
├── README.md
├── requirements.txt
├── paquete/
│   ├── subpaquete1/
│   │   ├── __init__.py
│   │   └── ejemplo.py
│   ├── subpaquete2/
│   │   ├── __init__.py
│   │   └── ejemplo.py
│   ├── __init__.py
│   ├── __main__.py
│   └── app.py
├── config.toml
├── tests/...
├── .venv/...
├── .git/...
└── .gitignore
```

Esta estructura **no es adecuada para distribuir código a clientes o empresas**. Sin embargo, permite un rápido despliegue en cualquier plataforma, facilidad para compartir el código y cumple con todos los requerimientos arriba expuestos. Además, con muy pocos pasos se puede convertir en una **aplicación distribuible mediante paquetes paquetes pre-compilados** (`wheels`). Por desgracia, esos conocimientos quedan fuera del alcance de este curso.

Archivos del proyecto.

`README.md`

El archivo `README.md` es un archivo de texto plano que se utiliza para proporcionar información sobre el proyecto en el que se está trabajando. El archivo README es una parte importante de cualquier proyecto de software ya que proporciona información sobre el propósito y la funcionalidad del proyecto, instrucciones sobre cómo instalar y utilizar el software, y otros detalles importantes para los usuarios y desarrolladores.

El archivo `README.md` a menudo se escribe en formato Markdown, que es un lenguaje de marcado ligero que permite agregar formatos de texto simples, como encabezados, listas y enlaces. El formato Markdown es muy popular para la documentación de proyectos de software porque es fácil de leer y escribir y se puede convertir fácilmente a otros formatos de documentación, como HTML o PDF.

`requirements.txt`

El archivo `requirements.txt` es un archivo de texto que se utiliza para listar las dependencias de un proyecto de Python. Cada línea del archivo especifica el nombre de un paquete y la versión específica que se requiere. El archivo se utiliza comúnmente para automatizar la instalación de todas las dependencias de un proyecto.

Cuando se trabaja en un proyecto de Python, es común que el proyecto dependa de muchos otros paquetes y bibliotecas de terceros. En lugar de instalar manualmente cada paquete, se puede utilizar el archivo `requirements.txt` para especificar todas las dependencias del proyecto. Esto permite una instalación rápida y fácil de todas las dependencias del proyecto a través del comando `pip install -r requirements.txt`.

El archivo `requirements.txt` debe estar en el directorio raíz del proyecto y cada línea debe seguir el siguiente formato:

```
nombre-del-paquete==versión
```

Por ejemplo, si el proyecto depende de la biblioteca `requests` en su versión `2.25.1`, la línea correspondiente en el archivo `requirements.txt` sería:

```
requests==2.25.1
```

Además de especificar las dependencias, el archivo `requirements.txt` también se puede utilizar para especificar otras opciones como paquetes adicionales para desarrollo o para especificar una fuente de distribución diferente a la fuente por defecto.

Para generar un archivo `requirements.txt` de forma automática con las dependencias de nuestro proyecto tenemos dos opciones:

- **Si estamos usando un entorno virtual**, en teoría solo habremos instalado los paquetes necesarios para el proyecto. Podemos generar el archivo `requirements.txt` con todas las dependencias del proyecto si hacemos con el entorno activado:

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

- Una opción más general, sobre todo si no estamos trabajando en un entorno virtual, es usar el módulo `pipreqs`, que analiza los archivos y extrae las dependencias de terceros:

```
$ python -m install pipreqs
$ pipreqs /ruta/carpeta/proyecto
```

Paquete.

La aplicación que se va a ejecutar estará contenida en un paquete. Dentro de la carpeta del paquete se encuentra el código de la aplicación organizado en subpaquetes y módulos, el archivo `__init__.py` para iniciar el paquete y el módulo `__main__.py` que servirá de punto de entrada al programa.

`config.toml`

TOML (Tom's Obvious, Minimal Language) es un formato de archivo de configuración creado para ser fácil de leer y escribir tanto para humanos como para máquinas. TOML se utiliza a menudo en aplicaciones y proyectos de Python para definir configuraciones, opciones y variables de entorno.

TOML es similar a otros formatos de archivo de configuración como YAML e INI, pero se diferencia por su sintaxis clara y estructura jerárquica. Los archivos TOML están formateados como una tabla que consiste en secciones y claves-valor. Las secciones se definen entre corchetes (`[]`) y las claves-valor se definen en pares, separados por un igual (`=`) o dos puntos (`:`).

Por ejemplo, si quisieramos hacer un archivo de configuración para un programa que accede a un servidor, podría ser algo así:

```
# Configuration file for my API

[server]
host = "pcasocieee.upct.es"
port = 50600

[auth]
username = "commodoro"
password = "123456"
```

Para acceder a los datos como un diccionario desde Python se puede utilizar el módulo `tomllib`, introducido en Python 3.11.

```
import tomllib

with open("config.toml", "rb") as f:  # ojo, 'rb'
    CONFIG = tomllib.load(f)

print('Host', CONFIG['server']['host'])
print('Puerto', CONFIG['server']['port'])
```

`CONFIG` es un diccionario con los datos contenidos en `config.toml`. Si hacemos esto en el módulo `__init__.py` tendremos acceso a la configuración desde todos los módulos y subpaquetes del paquete.

> El módulo `tomllib` no soporta la escritura de TOML. Para ello existen varios paquetes de terceros. Personalemente recomiendo `tomlkit`.

Tests.

La carpeta de tests contendrá módulos con diferentes pruebas accediendo a los módulos y paquetes de la aplicación. En este curso no se trabajá con ellos pero **es una herramienta muy importante**.

Entorno virtual.

Cuando empecemos un proyecto, es recomendable utilizar un entorno virtual para encapsular todas las dependecias y no afectar al entorno global. Puedes revisar el apartado de entornos virtuales para aprender a crear uno y gestionarlo.

Git.

El control de versiones con git es esencial en cualquier proyecto de software, incluyendo los proyectos de Python. Permite a los desarrolladores colaborar en el mismo código fuente, mantener un registro completo de las modificaciones realizadas en el código a lo largo del tiempo y recuperar versiones antiguas en caso de errores o necesidades específicas.

Además, el uso de git permite a trabajar en diferentes entornos (por ejemplo, diferentes máquinas) y mantener la misma versión del código fuente, lo que evita problemas de compatibilidad.

La carpeta `.git` se creará sola cuando iniciemos un repositorio de git (en VSCode por defecto ocultará esa carpeta en el explorador de archivos, ¡pero eso no quiere decir que no exista!). El archivo `.gitignore` es una lista de archivos y carpetas que no queremos que formen parte nunca del repositorio. Git simplemente ignorará su existencia. En proyectos de Python es recomendable incluir ahí las carpetas `__pycache__`, que es código precompilado que se produce a ejecutar los paquetes del proyecto y los archivos del entorno virtual. Ejemplo de `.gitignore` básico:

```
__pycache__
.venv
```

Pasos para crear la aplicación.

Para crear tu aplicación siguiendo el esquema descrito, sigue los siguientes pasos:

1. **Crea la estructura de carpetas del proyecto**.
2. **Crea un entorno virtual**. Y actívalo, claro.
3. **Crea el repositorio git**. Usando `git init` en la carpeta raíz del proyecto. Añade un remoto para subir el código a plataformas como GitHub.
4. **Desarrolla el programa.** Primero planifica y después escribe el código. Con **cada funcionalidad añadida haz un commit en `git`** para guardar los cambios.
5. **Testea**. Realiza las pruebas necesarias para comprobar que tu código funciona.

6. **Crea un README.md**. Donde expliques qué hace el programa, como interactuar con él y como instalarlo.
7. **Crea la lista de dependencias**. Genera tu archivo `requirements.txt` con uno de los dos métodos enseñados.

Una vez realizados estos pasos la aplicación está lista para ser compartida y distribuida para proyectos de desarrollo en cualquier sistema.

Instalar aplicación.

Una vez que tenemos la aplicación construida podemos distribuirla en formato `.zip` (obviando la carpeta .venv) o descargarla desde un repositorio remoto con `git clone {url}`.

Una vez descargada tendremos que crear un nuevo entorno virtual dentro de la carpeta del proyecto e instalar las dependencias mediante:

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

Ejecutar aplicación.

Para ejecutar la aplicación podemos hacerlo con el entorno virtual activado dentro de la carpeta del proyecto:

```
python -m paquete
```

> Recuerda, ¡la carpeta del proyecto no es la del paquete!

Si queremos lanzar el programa sin el entorno virtual activado deberemos llamar a la copia de Python en el entorno virtual:

```
.venv/bin/python -m paquete
```

Si no estamos en la carpeta del proyecto y queremos, por ejemplo, ejecutar el código de forma automática, debemos crear un script de Bash/PowerShell para ejecutarlo.

```
cd ruta/al/paquete
.venv/bin/python -m paquete
```

Crear un ejecutable.

Para desarrollos en Windows existen paquetes como `auto-py-to-exe` o `py2exe` con funcionalidades que permiten crear ejecutables (`.exe`) de Windows.