ChatGPT

Conceptos Fundamentales del Lenguaje Python

Conociendo Python

¿Qué es Python?
Python es un lenguaje de programación de alto nivel, interpretado y enfocado en la legibilidad del código [1]. Fue creado a finales de los años 80 por Guido van Rossum (publicado en 1991) con el objetivo de hacer la programación más accesible para todos, gracias a una sintaxis clara y sencilla [2]. Python es multiparadigma (soporta orientación a objetos, programación imperativa e incluso funcional) y de *propósito general*, lo que significa que se puede usar en una amplia variedad de dominios y aplicaciones [3]. Además, es un lenguaje *interpretado* (no requiere compilar el código manualmente antes de ejecutarlo) y *multiplataforma*, funcionando en sistemas operativos como Windows, Linux o macOS [1]. Python implementa *tipado dinámico*, es decir, no se declaran tipos de datos para las variables y un mismo nombre puede referirse a distintos tipos de objeto a lo largo del programa [4]. También es considerado un lenguaje **orientado a objetos**, dado que permite definir clases, crear instancias (objetos) y soporta conceptos de la POO como herencia y polimorfismo, aunque conserva flexibilidad para usar otros estilos (por eso se le cataloga multiparadigma).

Facilidad de uso y legibilidad: Una de las características más destacadas de Python es su énfasis en la legibilidad y simplicidad. Su sintaxis usa palabras en lugar de símbolos y se acerca al lenguaje natural, lo que reduce la curva de aprendizaje para principiantes [5] [6]. Por ejemplo, los bloques de código se definen por **indentación** (espacios o tabulaciones al inicio de la línea) en lugar de llaves `{}` u otras marcas; esto obliga a escribir un código ordenado y legible por convención. Python requiere muchas menos líneas de código para lograr una tarea en comparación con lenguajes como Java o C++, facilitando su uso y mantenimiento [7]. Esta sencillez, junto con una extensa documentación y una sintaxis coherente, hace que Python sea conocido por ser **fácil de aprender** y aplicar incluso para quienes dan sus primeros pasos en programación [2].

Aplicaciones y desarrollo web: Python es un lenguaje de propósito general, lo que implica que es utilizado en numerosos campos: desde **desarrollo web**, ciencia de datos e inteligencia artificial, hasta automatización de tareas, administración de sistemas, desarrollo de videojuegos y más. En el desarrollo de sitios web, Python ofrece frameworks populares como **Django** y Flask que permiten crear aplicaciones web de manera rápida y eficaz [8]. **Django** en particular es un framework web de alto nivel escrito en Python que facilita el desarrollo rápido de sitios web seguros y mantenibles, encargándose de muchos detalles de bajo nivel para que el programador pueda concentrarse en la lógica de la aplicación [9]. Gracias a herramientas como estas, Python se ha consolidado tanto en el backend de aplicaciones web como en servicios web (APIs), demostrando su versatilidad también en Internet.

Multipropósito y multiparadigma: Python está diseñado como un lenguaje **multipropósito**, es decir, no está limitado a un tipo específico de desarrollo. Con Python se puede desde *automatizar* tareas simples de script hasta construir complejos sistemas distribuidos. Asimismo, es **multiparadigma**: soporta programación orientada a objetos, programación estructurada imperativa y, en menor medida, programación funcional [1]. Por ejemplo, uno puede definir clases y objetos (POO) o simplemente escribir funciones sueltas y estructuras procedimentales, según convenga. Esta flexibilidad de paradigmas permite al desarrollador elegir el estilo que mejor se adapte al problema, o combinar estilos. En cualquier caso, Python fomenta un código claro y "*pitónico*", siguiendo principios como *"la*

legibilidad cuenta" y *"debe haber una –y preferiblemente solo una– forma obvia de hacerlo"* (parte del Zen of Python, la filosofía del lenguaje [10] [11]).

Python como lenguaje interpretado: Al ser un lenguaje interpretado, el código Python se ejecuta **mediante un intérprete** en tiempo de ejecución, sin necesidad de compilar a lenguaje máquina previamente. Esto acelera el ciclo de desarrollo, ya que puedes escribir y ejecutar código inmediatamente, probar cambios de forma interactiva (por ejemplo en la consola REPL) y usarlo de forma flexible en scripts. Internamente, la implementación estándar (CPython) compila el código fuente a un bytecode intermedio (.pyc) que luego interpreta, pero este proceso es transparente para el usuario [12]. En la práctica, significa que para ejecutar un programa Python basta con tener instalado el intérprete e invocar el archivo .py; el intérprete se encarga de traducir y ejecutar las instrucciones *sobre la marcha*. Esto contrasta con lenguajes compilados (como C++), donde se debe compilar todo el programa a binario antes de ejecutarlo. La naturaleza interpretada de Python facilita la experimentación y la ejecución parcial de código (por ejemplo, probando funciones en la consola interactiva), a costa de generalmente ser más lento en ejecución que un equivalente compilado, aunque existen optimizaciones como *Just-in-Time compilers* (PyPy) y extensiones en C para código crítico que mitigan este punto.

Portabilidad (multiplataforma): Python es **multiplataforma**, lo que significa que el mismo código Python puede ejecutarse sin cambios en distintos sistemas operativos (Windows, Linux/Unix, macOS, etc.) [13]. El intérprete de Python está disponible oficialmente para las principales plataformas, y la amplia mayoría de las bibliotecas estándar y de terceros son también portables. Esto simplifica el desarrollo, ya que un programa escrito en Python puede distribuirse a usuarios de diferentes entornos con mínimas adaptaciones. Por ejemplo, Python es muy usado en servidores Linux, pero también en entornos de escritorio Windows o Mac, e incluso en dispositivos móviles o embebidos (existen versiones como **MicroPython** para microcontroladores). La multiplataforma, junto con su distribución libre, ha contribuido a la enorme adopción global del lenguaje.

Otras características destacadas:
- **Libre y de código abierto:** Python es de **distribución libre** y código abierto, administrado por la Python Software Foundation. Esto implica que es gratuito para uso comercial o personal, y que su código fuente está disponible para la comunidad. La licencia de Python (Python Software Foundation License) permite su uso, modificación y distribución libremente [14].
- **Biblioteca estándar extensa:** Python sigue la filosofía de "*baterías incluidas*", ofreciendo una **extensa biblioteca estándar** con módulos para multitud de tareas: desde manejo de sistema operativo, operaciones matemáticas, manipulación de texto, Internet, hasta interfaces gráficas, bases de datos, etc. Esta rica biblioteca estándar permite resolver muchos problemas sin instalar nada adicional [15]. Por ejemplo, incluye módulos para HTTP, manejo de JSON, expresiones regulares, concurrencia, acceso a archivos, entre otros, lo que ahorra tiempo y esfuerzo al programador. Además, su ecosistema de paquetes externos (disponibles via pip) amplía aún más sus capacidades.
- **Soporte para bases de datos:** Python puede conectarse y trabajar con múltiples sistemas de **bases de datos** populares. Existen interfaces o librerías para SQL (p.ej. SQLite viene integrada, y hay conectores para PostgreSQL, MySQL, Oracle, SQL Server) e incluso para bases NoSQL. La especificación Python DB-API unifica en lo posible el uso de bases de datos relacionales en Python, de forma que un mismo programa pueda soportar distintos motores simplemente cambiando el módulo de conexión [16]. Asimismo, frameworks como SQLAlchemy proporcionan un ORM poderoso que facilita el trabajo con varias bases de datos de forma transparente.
- **Gran comunidad y soporte:** Python cuenta con una **comunidad muy numerosa y activa** a nivel mundial [17]. Esto se traduce en abundante documentación, foros (Stack Overflow, Reddit, etc.), tutoriales, cursos, conferencias (PyCon) y grupos de usuarios en prácticamente cualquier país e idioma. La comunidad contribuye con miles de librerías de código abierto, soluciona dudas y mejora

continuamente el ecosistema. Gracias a ello, es fácil encontrar ayuda y recursos para aprender Python o resolver problemas, lo que disminuye la barrera de entrada. La popularidad de Python ha crecido tanto que hoy es uno de los lenguajes más utilizados en campos como ciencia de datos, aprendizaje automático, automatización de infraestructuras, desarrollo web y más.

En resumen, Python es un lenguaje interpretado, multiparadigma y multipropósito, conocido por su sencillez y legibilidad, con un amplio soporte de bibliotecas y una comunidad vibrante. Estas cualidades fundamentales explican por qué Python es a la vez **fácil de aprender** para principiantes y **potente** para expertos al abordar proyectos de todo tamaño.

Entorno de Ejecución de Python

Instalación de Python: Para comenzar a programar en Python es necesario instalar el intérprete. Python se distribuye oficialmente desde el sitio python.org con instaladores para Windows y macOS, y en muchas distribuciones Linux viene preinstalado o se puede instalar fácilmente desde los repositorios del sistema [18] [19]. En Windows, el instalador oficial incluye opcionalmente añadir Python al PATH del sistema para poder usarlo desde la consola. En macOS se puede usar el instalador .pkg oficial (o gestores de paquetes como Homebrew), y en Linux normalmente se instala vía el gestor de paquetes (apt, dnf, pacman, etc.) si no está ya incluido. Es importante destacar que a partir de Python 3.10+ solo se mantiene la rama Python 3 (Python 2 ya está obsoleto); hoy en 2025 la versión estable más reciente es Python 3.14. Tras instalar Python, se dispone tanto del **intérprete interactivo** como de la herramienta de **gestión de paquetes pip** para extender su funcionalidad con módulos externos.

La consola de comandos e intérprete interactivo: Una vez instalado, Python se puede utilizar mediante la **consola de comandos** del sistema operativo. En entornos Unix (Linux/Mac) se abre una terminal y en Windows se puede usar el Símbolo del Sistema o PowerShell. Ejecutando el comando `python` (o `python3` en sistemas donde Python 2 está presente) se inicia el **intérprete interactivo** de Python. En este modo interactivo (REPL: *Read-Eval-Print Loop*), veremos aparecer el prompt `>>>` donde podemos escribir instrucciones Python y obtener el resultado inmediatamente. Por ejemplo: `>>> print("Hola")` imprimirá `Hola` en la siguiente línea. Este entorno es muy útil para probar código rápidamente, calcular expresiones o depurar interactivamente. Alternativamente, se puede invocar un script de Python desde la consola indicando el nombre del archivo. Por ejemplo, suponiendo un archivo `prueba.py` en el directorio actual, los siguientes comandos son equivalentes en Windows para ejecutarlo:

```
C:\> **python prueba.py**
C:\> **py prueba.py**
```

Ambos lanzarán el intérprete para ejecutar el archivo especificado [20]. En sistemas *Unix* sería análogo usando `python3 prueba.py` desde la terminal. Asimismo, en Windows es posible **ejecutar un archivo .py con doble clic** desde el Explorador; esto abrirá una ventana de terminal que correrá el script y se cerrará al finalizar. Por defecto, al terminar la ejecución la ventana se cierra inmediatamente, por lo que a veces se añade al final del script algo como `input()` para pausar el cierre hasta que el usuario presione Enter.

Archivos fuente .py y .pyw: Los programas Python se almacenan típicamente en archivos de texto con extensión `.py` (por ejemplo `mi_programa.py`). Estos archivos contienen código fuente Python y son **módulos** ejecutables por el intérprete. En entornos Windows existe además la extensión `.pyw`, asociada a **pythonw.exe**, pensada para scripts de interfaz gráfica que no necesitan consola. La

diferencia es que un archivo `.py` al ejecutarse abrirá una ventana de terminal (la consola) vinculada al programa, mientras que un `.pyw` se ejecuta *sin* mostrar ninguna ventana de consola [21]. En otras palabras, `python.exe` (usado para .py) lanza una consola para entrada/salida estándar, mientras que `pythonw.exe` (usado para .pyw) suprime la consola. Esto es útil para aplicaciones de GUI donde una ventana de terminal vacía sería molesta. Por ejemplo, un script Tcl/Tk o PyQt podría guardarse como .pyw para que al hacer doble clic solo aparezca la ventana gráfica de la aplicación. Hay que usar `.pyw` **solo** cuando el programa *no* necesita interactuar por consola (pedir datos o imprimir mensajes); en caso contrario, conviene usar `.py` para disponer de la terminal.

Sincronía de la ejecución: Relacionado con lo anterior, existe una diferencia de **ejecución síncrona vs asíncrona** entre `python.exe` y `pythonw.exe` en Windows. Cuando ejecutamos un script con `python.exe` (o desde una terminal), el proceso es *síncrono*: la consola queda ocupada hasta que el programa finaliza, y no podemos lanzar otro programa Python en la misma terminal hasta entonces. En cambio, `pythonw.exe` ejecuta el script de forma *asíncrona*, retornando inmediatamente el control a la consola (o al explorador de archivos) mientras el programa corre en segundo plano [22]. Esto permite, por ejemplo, abrir varias aplicaciones Python con interfaz gráfica simultáneamente desde el entorno gráfico. En entornos Linux/Unix, la distinción no es exactamente igual (normalmente se maneja ejecutando en background con `&`), pero el concepto importante es que los scripts .pyw en Windows no bloquean una terminal. Para la mayoría de casos de uso diario, esta diferencia no afecta más que al comportamiento al lanzar programas desde el explorador Windows.

Entorno integrado y herramientas: Python también provee entornos simples de desarrollo integrados. Al instalar Python normalmente se incluye **IDLE**, un entorno básico con editor de texto coloreado y una ventana de intérprete interactivo, útil para escribir pequeños programas si no se tiene otro editor. No obstante, para proyectos más grandes suelen emplearse editores de código o IDEs más robustos (como VSCode, PyCharm, etc.), los cuales requieren haber instalado Python pero ofrecen funcionalidades avanzadas (autocompletado, depuración, etc.). Otra herramienta útil es **IPython** (un intérprete mejorado) y los **Jupyter Notebooks**, muy populares en ciencia de datos, que permiten combinar código Python con texto y visualizaciones en un entorno de cuaderno interactivo.

El administrador de paquetes pip: Una de las grandes ventajas de Python es su ecosistema de paquetes externos. Python incluye de fábrica el gestor de paquetes **pip** (Python Package Installer), con el cual es posible **instalar, actualizar o desinstalar** librerías de terceros de forma sencilla. Pip trabaja en línea con el repositorio PyPI (Python Package Index), que centraliza decenas de miles de paquetes disponibles. Para usar pip, se ejecuta en la consola el comando `pip install nombre_paquete`. Por ejemplo, `pip install requests` descargaría e instalaría la biblioteca Requests (para hacer peticiones HTTP) automáticamente. Este gestor resuelve dependencias y facilita compartir entornos reproducibles mediante un archivo de requerimientos. A partir de Python 3.4, pip viene incluido por defecto, aunque en versiones anteriores debía instalarse manualmente. Un uso típico es:

```
$ **python -m pip install nombre_paquete**
```

que buscará la última versión del paquete en PyPI e instalará sus dependencias necesarias [23]. Pip también permite instalar una versión específica (`pip install paquete==2.0`), listar paquetes instalados (`pip list`), actualizar (`pip install --upgrade paquete`) o desinstalar (`pip uninstall paquete`). Para aislar paquetes por proyecto y evitar conflictos, Python ofrece **entornos virtuales** (con `python -m venv`), de modo que pip instale las librerías localmente en ese entorno en lugar de globalmente. En resumen, pip es la herramienta estándar para gestionar las extensiones de

Python, contribuyendo a la modularidad y riqueza del lenguaje al acceder fácilmente a frameworks web, herramientas de ciencia de datos, utilidades de sistema y mucho más.

Conceptos Básicos de Python

Pasemos ahora a revisar algunos **conceptos básicos del lenguaje** en cuanto a sintaxis y construcción de programas.

La instrucción `print`

En Python 3, `print()` es una función integrada que sirve para **mostrar datos en la salida estándar** (normalmente la pantalla o consola). Se usa para imprimir texto, resultados de expresiones, valores de variables, etc. Por ejemplo:

```
print("Hola, mundo")
```

mostrará en pantalla la frase **Hola, mundo**. La función `print` admite múltiples argumentos separados por comas –los concatenará con espacios por defecto– y diversos parámetros opcionales para formatear la salida. Por **defecto**, `print()` termina cada llamada con un salto de línea (nuevo renglón) al final [24]. Así, dos llamadas consecutivas a print imprimirán cada resultado en una línea separada. Si se desea cambiar este comportamiento (por ejemplo, para que no haya salto de línea), puede usarse el parámetro `end`: `print("Hola", end="")` imprimirá *Hola* **sin** salto final, de forma que lo siguiente que se imprima quede en la misma línea. Igualmente, el separador entre múltiples argumentos por defecto es un espacio, pero puede modificarse con el parámetro `sep`. Por ejemplo, `print("A", "B", sep="--")` mostrará `A--B`.

Python soporta además **f-strings** (cadenas formateadas, disponibles desde Python 3.6) que facilitan la inserción de variables dentro de cadenas para imprimirlas. Ejemplo:

```
nombre = "Ana"
edad = 30
print(f"Me llamo {nombre} y tengo {edad} años.")
```

Esto imprimiría: *Me llamo Ana y tengo 30 años.* Las *f-strings* permiten escribir expresiones entre `{}` dentro de la cadena, las cuales Python evaluará y reemplazará por su valor en la salida, haciendo muy conveniente formatear mensajes [25] [26]. En resumen, `print()` es una herramienta fundamental para verificar el funcionamiento del código (debugging sencillo) y para interactuar mostrando resultados al usuario.

Operaciones aritméticas básicas

Python puede usarse como una potente calculadora. Soporta las **cuatro operaciones aritméticas básicas**: **suma** (`+`), **resta** (`-`), **multiplicación** (`*`) y **división** (`/`) [27]. Estas operaciones se pueden realizar con números enteros, de coma flotante, complejos, etc. Algunas consideraciones importantes: al sumar, restar o multiplicar **enteros**, el resultado es un entero (ejemplo: `5 * 6` da `30`); pero al realizar cualquier operación con números **decimales (float)**, o una división incluso entre enteros, el resultado será *flotante* (con parte decimal) [28] [29]. En Python 3, la división `/` siempre produce un número de punto flotante, aun cuando el resultado matemático sea entero (por ejemplo `9/3` resulta

en `3.0` y no en `3`) [30]. Para obtener el resultado entero de una división (descartando el resto) se utiliza el **operador de división entera** `//` [31]. Por ejemplo, `11 // 3` produce `3` (cociente entero) y `11 % 3` produce `2` (el **resto** de dividir 11 entre 3) [32]. El operador **módulo** `%` es útil para obtener el residuo de divisiones, determinar si un número es divisible por otro (si `a % b == 0` entonces *a* es divisible por *b*), etc. Python también soporta la **potenciación** con el operador `**` [33]. Por ejemplo `2 ** 3` da `8` (2 al cubo) y `10 ** 0.5` da `3.162277...` (raíz cuadrada de 10, ya que 0.5 es 1/2). Además, existen funciones matemáticas avanzadas en el módulo estándar `math` (como `math.sqrt()` para raíz cuadrada, `math.sin`, logaritmos, etc.) y tipos numéricos especiales (decimal con precisión fija, fracciones, números complejos con `j`, etc.), pero en cuanto a operadores básicos, los mencionados cubren la mayoría de necesidades aritméticas.

El orden de precedencia de operadores en Python es el usual de matemáticas: las **potencias** se calculan primero, luego *multiplicación*, *división*, *división entera* y *módulo*, y finalmente *suma* y *resta*. Se puede usar paréntesis `( )` para forzar un orden específico cuando sea necesario. Por ejemplo: `1 + 2 * 3` resulta en `7` (porque multiplica 2*3 antes de sumar 1), mientras que `(1 + 2) * 3` resulta en `9`. *Cabe señalar que Python arrojará un error de tipo* ZeroDivisionError* si se intenta dividir entre cero (`a/0` o `a//0`), ya que esta operación no está definida [34].

Sintaxis básica del lenguaje

La sintaxis de Python tiene algunas reglas y particularidades que la distinguen de otros lenguajes:

- **Nombres de variables e identificadores:** En Python, los identificadores (nombres de variables, funciones, clases, etc.) **diferencian mayúsculas de minúsculas** (es *case-sensitive*). Por convención se usan minúsculas para variables y funciones, y CamelCase para nombres de clases, pero a nivel del lenguaje `variable`, `Variable` y `VARIABLE` son tres identificadores distintos. Los nombres válidos pueden contener letras (A-Za-z), dígitos y el caracter `_` (guión bajo), **pero no pueden comenzar con un dígito** [35]. Es decir, `valor1` o `_tmp` son nombres válidos, mientras que `1valor` no lo es (causaría un error de sintaxis). Asimismo, no se permiten espacios ni caracteres especiales como `ñ`, `$`, `-` etc. en los nombres. Python tampoco permite usar como identificadores las **palabras reservadas** del lenguaje (palabras clave de la sintaxis como `for`, `while`, `if`, `True`, `None`, `class`, etc.) [35]; intentar asignar `True = 5` daría un error de sintaxis porque `True` es una constante booleana reservada. Otra buena práctica es evitar **ocultar nombres integrados** de Python –por ejemplo no llames a una variable `list` o `sum`, porque sobreescribirías las funciones built-in `list()` o `sum()` y podrías causar comportamientos inesperados [36]. Python no requiere declarar las variables antes de usarlas; basta asignarles un valor para que existan en el ámbito actual. El tipo de dato de una variable viene dado por el objeto que almacena, y puede cambiar dinámicamente: una variable puede empezar valiendo un entero y luego puedo asignarle una cadena, sin error (aunque hacerlo deliberadamente no es recomendable por legibilidad). En resumen: elige nombres descriptivos, usando solo caracteres permitidos, y sigue las convenciones de estilo (*snake_case* para variables y funciones, *CapWords* para clases) para escribir un código claro [37].

- **Funciones y métodos:** En Python se definen **funciones** usando la palabra clave `def`. Por ejemplo:

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

define una función `suma` que toma dos parámetros y devuelve su suma. Las funciones se invocan escribiendo su nombre seguido de paréntesis con los argumentos: `suma(5, 7)` retornaría 12. Ahora bien, cuando una función se define dentro de una clase, pasa a llamarse **método**. Es decir, en programación orientada a objetos, las *funciones asociadas a un objeto o clase* se conocen como métodos. En Python, todas las funciones definidas en el cuerpo de una clase son métodos de esa clase (instancias de `function` que, al accederse vía un objeto, se convierten en métodos *bound* a ese objeto). Por ejemplo:

```
class Persona:
    def __init__(self, nombre):
        self.nombre = nombre
    def saludar(self):
        print(f"Hola, soy {self.nombre}")
```

Aquí `saludar` es un método de la clase `Persona`. Podemos crear una instancia: `p = Persona("Ana")` y luego llamar `p.saludar()`, lo cual imprimirá *Hola, soy Ana*. Nótese que los métodos en Python siempre llevan `self` como primer parámetro (por convención) para referirse a la instancia sobre la que operan. Cuando llamamos `p.saludar()`, Python pasa implícitamente `self=p` al método. En general, **cualquier función definida en una clase se denomina método** (incluso los métodos especiales como `__init__`). Por otro lado, también existen **métodos de objetos integrados**: por ejemplo, las cadenas de texto tienen métodos como `.upper()` (que devuelve la cadena en mayúsculas) o las listas tienen `.append()` (para agregar elementos). Estos métodos se invocan con la sintaxis de *dot notation* (punto): `mi_cadena.upper()` o `mi_lista.append(5)`. En conclusión, *método* no es más que un término para referirnos a una función asociada a un objeto; en Python no hay una distinción de sintaxis fuerte entre funciones "globales" y métodos (ambas se definen con `def`), salvo que los métodos pertenecen a clases/objetos y usualmente operan sobre `self`. (Como detalle, Python no usa el término "procedimiento": toda función que no retorne nada explícitamente en realidad devuelve `None` por defecto).

- **Indentación y bloques de código:** A diferencia de la mayoría de lenguajes, Python **utiliza la indentación (sangría) para definir la estructura de bloques** en el código [38]. Esto significa que **no existen llaves** `{}` para delimitar el inicio y fin de funciones, bucles, condicionales, etc. En su lugar, el nivel de sangrado (número de espacios o tabs al comienzo de la línea) determina qué instrucciones están dentro de un bloque. Por convención se usan 4 espacios por nivel de indentación (el intérprete también acepta tabuladores, pero se recomienda espacios para evitar inconsistencias). Ejemplo:

```
if x > 0:
    print("x es positivo")
    x = x - 1
    print("Decrementando x")
print("Esto se ejecuta siempre")
```

En este código, las tres líneas indentadas bajo el `if` forman parte del bloque condicional, que solo se ejecutará si `x>0`. La última línea no está indentada, por lo que está fuera del `if` y se ejecutará independientemente. La indentación en Python es **obligatoria y significativa**: un error común es olvidarse de indentar o usar un nivel incorrecto, lo que produce errores de sintaxis o lógica. Por ejemplo, escribir una función sin indentación en su cuerpo es un error de

sintaxis. También es importante ser consistente: no se debe mezclar tabulaciones y espacios en la indentación de un mismo archivo, y todos los bloques anidados deben aumentar la sangría uniformemente. La ventaja de este mecanismo es que fuerza un código ordenado y legible; la estructura visual del código *es* la estructura lógica del programa. Muchos editores de texto ayudan manejando la indentación automáticamente después de dos puntos `:` (que en Python indican el comienzo de un bloque, por ejemplo tras `def`, `if`, `for`, etc.).

- **Comentarios:** Los comentarios son textos en el código fuente que el intérprete ignora, útiles para explicar o anotar partes del código. En Python, un comentario de línea se indica con el caracter `#`. **Todo lo que aparezca a la derecha de un `#` en una línea es ignorado** por el intérprete [39]. Por ejemplo:

```
total = precio * cantidad  # Cálculo del total a pagar
# El siguiente bloque actualiza el stock:
actualizar_inventario(producto, -cantidad)
```

En la primera línea, el comentario explica la operación realizada. En la segunda línea, toda la línea comienza con `#`, por lo que es un comentario completo (comentario de bloque) y Python no ejecutará nada allí. No existe una sintaxis especial para comentarios multilínea; si se desea comentar varias líneas, se puede anteponer `#` en cada una de ellas, o usar una cadena multilínea sin asignar a ninguna variable (esta técnica se aprovecha porque una *string* solitaria se interpreta como un literal no usado y se ignora en tiempo de ejecución) [40] [41]. Por ejemplo:

```
"""
Este es un comentario
de varias líneas usando
una cadena literal.
"""
```

La convención es utilizar este tipo de *docstrings* (strings con triple comilla `"""`) al inicio de módulos, funciones o clases para proporcionar documentación, ya que permanecen accesibles vía la función `help()` y no son eliminados por completo. Sin embargo, para comentar bloques de código temporalmente, lo más común es usar `#` en cada línea (muchos editores permiten comentar/descomentar bloques fácilmente). Es importante comentar el código *cuando sea necesario* para aclarar intenciones, pero evitando comentarios redundantes que describan cosas obvias. Asimismo, Python cuenta con la función `help()` y la documentación *docstring* para funciones/clases, lo que complementa los comentarios lineales tradicionales.

Manejo de módulos

En Python, un **módulo** es simplemente un archivo `.py` que contiene código (ya sean variables, funciones, clases, etc.). Los módulos permiten organizar el programa en varios archivos y reutilizar código fácilmente. Veamos las nociones básicas sobre módulos:

- **Importación de módulos:** Para usar el código de un módulo en otro (o en la sesión interactiva), Python proporciona la instrucción `import`. Usar `import modulo` hará que Python busque el archivo `modulo.py`, lo ejecute (definiendo todo su contenido) y disponibilice ese módulo en nuestro programa [42]. Por ejemplo, si tenemos un archivo `utilidades.py` con funciones

útiles, podemos importarlo con `import utilidades`. A partir de ese momento podremos usar sus funciones prefixando el nombre del módulo: p. ej. `utilidades.mi_funcion()`. También es común la sintaxis `from modulo import nombre` que **importa un símbolo específico** de un módulo directamente al espacio de nombres actual [43]. Por ejemplo: `from math import pi` nos permite usar `pi` directamente en lugar de `math.pi`. Incluso se puede importar *todos* los nombres públicos de un módulo con `from modulo import *`, pero esta práctica está **desaconsejada** ya que puede causar confusiones de nombres y reduce la legibilidad [44] (solo debería usarse en sesiones interactivas o casos muy controlados). Una variante útil es **asignar un alias** al módulo: `import numpy as np` permite referirse a `numpy` simplemente como `np` [45]. Esto se suele hacer con módulos de nombres largos o muy usados (por convención, por ejemplo, `import pandas as pd`). Es importante saber que cuando se importa un módulo, Python ejecuta su código *una sola vez* (la primera vez) y lo almacena en caché; si importamos el mismo módulo nuevamente en otro lugar, no se vuelve a ejecutar su código sino que se reutiliza el módulo ya cargado [46]. Si por alguna razón modificamos el módulo externo y queremos recargarlo sin reiniciar el intérprete, podemos usar `importlib.reload(modulo)`.

- **Módulos ejecutados como scripts:** Un mismo archivo Python puede usarse tanto como módulo **importable** como ejecutarse directamente como **script principal**. Python define una variable especial `__name__` en cada módulo: si el módulo se está ejecutando directamente (por ejemplo con `python modulo.py`), entonces `__name__ == "__main__"`, en cambio si el módulo fue importado, `__name__` toma el nombre del módulo. Esto nos permite escribir código condicional dentro del archivo para que solo se ejecute en caso de ser el programa principal. La típica plantilla es:

```
def funciones_y_clases():
    ...

# Código de pruebas o ejecución
if __name__ == "__main__":
    # Este bloque solo corre al ejecutar este archivo directamente
    prueba_funciones()
    ...
```

De esta manera, si hacemos `python modulo.py`, el bloque bajo `if __main__` correrá (por ejemplo realizando pruebas o acciones principales), pero si hacemos `import modulo` desde otro script, ese bloque no se ejecutará [47]. Esto es muy útil para incluir *demostraciones* o *tests simples* en el mismo módulo sin que interfieran cuando el módulo es importado. Muchos módulos de la biblioteca estándar utilizan este mecanismo. Por ejemplo, el módulo `math` no tendría tal bloque (pues no es un programa ejecutable por sí mismo), mientras que módulos diseñados para poder correr solos sí lo incluyen. En resumen, `if __name__ == "__main__":` es la forma en Python de indicar "ejecutar lo siguiente solo si este archivo es el programa principal".

- **Ruta de búsqueda de módulos:** Cuando hacemos un `import nombre_modulo`, el intérprete realiza una búsqueda para encontrar el módulo. Primero revisa si es un **módulo integrado** (built-in, escrito en C e incluido con Python) cuyo nombre coincida [48]. Si no, entonces busca un archivo `nombre_modulo.py` en los directorios listados en la variable `sys.path`. Esta ruta de búsqueda (`sys.path`) se inicializa al arrancar Python e incluye, en orden: **1)** el directorio actual

(o el directorio del script principal en ejecución), **2)** las rutas listadas en la variable de entorno **PYTHONPATH** (si está definida, similar a PATH del sistema, separando múltiples rutas), y **3)** los directorios estándar de instalación de Python (por ejemplo el paquete `site-packages` donde van módulos de terceros instalados) [48] [49]. En sistemas Windows, también incluye el directorio de instalación de Python. Por ejemplo, si tenemos un proyecto en `/home/usuario/proyecto/` y dentro un archivo `util.py`, al hacer `import util` desde un script en el mismo directorio, Python lo encontrará inmediatamente (directorio actual). Si `util.py` estuviera en otra ruta no estándar, tendríamos que añadir manualmente esa ruta a `sys.path` o usar PYTHONPATH para que Python lo encuentre. Cabe mencionar que los nombres de módulos son el nombre de archivo sin la extensión `.py`. Además, Python almacena los módulos ya importados en `sys.modules` para no cargarlos dos veces. La búsqueda de módulos puede personalizarse en casos avanzados mediante import hooks, pero típicamente basta con asegurarse de que el módulo esté en la ruta.

- **Archivos compilados (`.pyc`):** Para acelerar la carga de módulos, Python automáticamente **compila** los archivos `.py` a **bytecode** la primera vez que se importan, generando archivos `.pyc` (Python compiled) almacenados en un directorio especial `__pycache__`. Por ejemplo, si importamos un módulo `spam` en Python 3.11, se creará un archivo `__pycache__/spam.cpython-311.pyc` [12]. La próxima vez que se importe `spam`, si el `.pyc` existe y está al día (no más antiguo que el `.py` fuente), Python cargará directamente el bytecode, evitando recompilar el módulo, lo cual mejora el rendimiento. Estos archivos `.pyc` son específicos de la versión de Python (de ahí el sufijo `cpython-33`, `cpython-310`, etc. según versión) y de la plataforma en ciertos casos. Normalmente no tenemos que preocuparnos por ellos; Python se encarga de crearlos, usarlos y actualizarlos cuando el código fuente cambia. Si un módulo se ejecuta directamente como script, Python no genera .pyc en cache a menos que se importe posteriormente. También, si el archivo .py fuente no está presente (por ejemplo, distribuimos solo el .pyc), Python podrá usar el bytecode compilado directamente siempre que sea compatible con la versión. En síntesis, los archivos compilados son un detalle de implementación que agiliza importaciones, pero el programador típicamente sigue importando módulos de la misma forma sin notar la diferencia. (En caso de eliminar manualmente archivos .pyc o la carpeta `__pycache__`, Python simplemente volverá a compilar los .py cuando sea necesario).

- **La función `dir()`:** Python proporciona la función built-in `dir()` para **inspeccionar el contenido de módulos u objetos**. Si llamamos `dir(modulo)` obtendremos una lista de los nombres definidos en ese módulo (funciones, clases, variables, etc.) [50]. Por ejemplo, tras `import math`, ejecutar `dir(math)` listará algo como `['acos', 'acosh', 'asin', ... 'pi', 'e']` mostrando los nombres disponibles en el módulo `math` (constantes pi, e, y funciones trigonométricas, logarítmicas, etc.). Esto es útil para explorar módulos desconocidos. Asimismo, `dir()` sin argumentos devuelve los nombres presentes en el **ámbito actual** (local) [51], permitiendo ver qué variables hemos definido hasta ahora, qué módulos se han importado, etc. Cabe destacar que `dir()` es más informativa que otra cosa; para documentación más detallada se suele usar `help(nombre)` que muestra docstrings. `dir()` no listará los nombres internos que comienzan con guión bajo doble (a menos que se usen en el módulo), ni los nombres built-in a menos que se aplique sobre `builtins` explícitamente [52]. Pero para un vistazo rápido de contenido es muy práctica.

Manejo de paquetes

Un **paquete** en Python es un conjunto de módulos organizados en directorios. Técnicamente, un paquete es un módulo "especial" que puede contener submódulos o subpaquetes. Los paquetes

permiten estructurar proyectos grandes en múltiples niveles (similar a un árbol de directorios con archivos). Por ejemplo, podríamos tener un paquete `miaplicacion` con subpaquetes `miaplicacion.utils`, `miaplicacion.modelos`, etc., cada uno con sus módulos correspondientes. Para crear un paquete en Python tradicionalmente se incluía un archivo `__init__.py` vacío en la carpeta, indicando que dicho directorio es un paquete Python (en Python 3.3+ esto no es estrictamente necesario para *namespace packages*, pero sigue siendo común). Veamos algunos aspectos importantes:

- **Importación de paquetes:** Importar un paquete realmente importa el módulo `__init__.py` del paquete. Por ejemplo, si tenemos un paquete llamado `mis_utiles` con un submódulo `calculos.py` adentro, podemos hacer `import mis_utiles.calculos`. Esto importará primero `mis_utiles` (ejecutando su `__init__.py`) y luego el submódulo `calculos`. Podemos entonces usar `mis_utiles.calculos.funcionX()`. También es válido `from mis_utiles import calculos` para traer el submódulo directamente [53]. Si queremos importar *todo* un paquete (todos sus submódulos), tendríamos que importarlos individualmente o utilizar técnicas como `import pkgutil` para descubrimiento dinámico – pero típicamente se importa solo lo necesario. Un paquete puede definir en `__init__.py` una lista `__all__` para controlar qué submódulos se importan con un `from paquete import *`. Pero como se mencionó, el import * no se recomienda en código de producción. En resumen, la sintaxis de importación soporta el acceso jerárquico con puntos: `package.subpackage.module`. Python buscará en su ruta de módulos un directorio `package` que contenga ese submódulo.

- **Referencias internas (importes relativos):** Cuando se está trabajando *dentro* de un paquete, a veces conviene importar módulos relativos al propio paquete sin repetir todo el camino. Python permite **imports relativos** usando la sintaxis de punto. Por ejemplo, imaginemos un paquete `sonido` con subpaquetes `efectos` y `filtros`. Si dentro de `sonido/filtros/vocoder.py` necesitamos importar el módulo hermano `sonido/efectos/echo.py`, podemos escribir: `from sound.effects import echo` (import absoluto) *o* hacer un import relativo como `from ..effects import echo` [53] [54]. Aquí `..` indica "subir un nivel" (al paquete padre `sonido`) y luego entrar al subpaquete `effects`. De igual manera, `from . import echo` dentro de `effects/__init__.py` importaría un submódulo `echo` del mismo paquete. Los imports relativos usan uno o más `.` al inicio del nombre: uno `.` refiere al paquete actual, `..` al padre, `...` dos niveles arriba, etc. Es importante notar que esta sintaxis *solo funciona dentro de módulos que forman parte de un paquete* – el intérprete necesita conocer el atributo `__package__` del módulo. Si intentamos ejecutar un módulo que use imports relativos directamente, puede fallar porque el entorno no sabe que pertenece a un paquete. Por eso, los **scripts principales deben usar imports absolutos**, y los imports relativos se reservan para módulos internos de paquetes [54] [55].

- **Paquetes en múltiples directorios:** Python soporta la idea de *namespace packages* que permiten que el contenido de un paquete se distribuya en varios directorios diferentes. Esto se logra mediante el manejo del atributo especial `__path__` en los paquetes. Al importar un paquete, Python inicializa `package.__path__` con una lista de directorios donde buscar submódulos [56]. Normalmente contiene solo el directorio del paquete, pero se puede extender programáticamente (o mediante la instalación de paquetes *distributed*) para abarcar varios lugares. Por ejemplo, podríamos tener una parte del paquete `miapp` instalada en `/usr/lib/miapp/` y otra parte en `/usr/local/miapp_plugins/`, y configurar `miapp.__path__` para incluir ambos. Entonces un `import miapp.moduloX` buscaría en ambos directorios. Esta característica se usa para *plugins* o extensiones modulares, pero no es muy común en desarrollos estándar. Desde Python 3.3, si faltan archivos `__init__.py`, Python trata ciertos

directorios como namespace packages automáticamente, permitiendo esta composición múltiple. En resumen, **un paquete puede abarcar varios directorios** si se configura adecuadamente su ruta de búsqueda (`__path__`), aunque típicamente cada paquete reside en una única carpeta del proyecto. Para la mayoría de desarrolladores, saber que esto es posible es suficiente; implementar namespace packages manualmente es raro.

En conclusión, los **paquetes** son simplemente módulos jerarquizados en directorios, útiles para organizar el código a gran escala. Se importan usando la notación de punto, pueden tener imports relativos para facilitar referencias internas, y ofrecen una estructura limpia para proyectos complejos (por ejemplo, las librerías estándar y de terceros usan paquetes extensivamente para agrupar submódulos de funcionalidad). Entender módulos y paquetes permite construir programas Python modulares, mantenibles y escalables, aprovechando todo el ecosistema de módulos disponibles.

Cada uno de estos conceptos –desde la sintaxis básica hasta la organización en paquetes– forma parte de los *fundamentos de Python*. Dominarlos sienta las bases para escribir programas correctos y idiomáticos en este lenguaje, y para aprovechar su potencia con buenas prácticas de desarrollo. Python combina facilidad de uso con versatilidad, lo que explica su adopción masiva tanto en entornos académicos como industriales, y conocer sus fundamentos es el primer paso para utilizarlo eficazmente en cualquier tipo de proyecto.

---

1 13 15 Python - Wikipedia, la enciclopedia libre
https://es.wikipedia.org/wiki/Python

2 3 8 Qué es Python: Guía Completa para Principiantes [2025] + Ejemplos Prácticos - Programación en Python
https://pythones.net/que-es-python-lenguaje-de-programacion/

4 18 19 38 1.2. Características — Materiales del entrenamiento de programación en Python - Nivel básico
https://entrenamiento-python-basico.readthedocs.io/es/2.7/leccion1/caracteristicas.html

5 6 7 10 11 14 17 ¿Por qué escoger el lenguaje python? Te lo contamos aquí
https://www.tokioschool.com/noticias/por-que-escoger-lenguaje-python/

9 Introducción a Django - Aprende desarrollo web | MDN
https://developer.mozilla.org/es/docs/Learn_web_development/Extensions/Server-side/Django/Introduction

12 43 44 45 46 47 48 49 50 51 52 53 54 55 56 6. Módulos — documentación de Python - 3.13.9
https://docs.python.org/es/3.13/tutorial/modules.html

16 DB-API - Qué es y cómo funciona - Recursos Python
https://recursospython.com/guias-y-manuales/python-db-api-que-es-y-como-funciona/

20 21 22 Ejecutar programas. Python. Bartolomé Sintes Marco. www.mclibre.org
https://www.mclibre.org/consultar/python/otros/python-uso.html

23 Instalando módulos de Python — documentación de Python - 3.14.0
https://docs.python.org/es/3/installing/index.html

24 25 26 Salida por pantalla: print(). Python. Bartolomé Sintes Marco. www.mclibre.org
https://www.mclibre.org/consultar/python/lecciones/python-salida-pantalla.html

27 28 29 30 31 32 33 34 Números y operaciones aritméticas elementales. Python. Bartolomé Sintes Marco. www.mclibre.org
https://www.mclibre.org/consultar/python/lecciones/python-operaciones-matematicas.html

35 36 37 El ámbito de las variables en Python y la regla `LEGB` explicados | DataCamp
https://www.datacamp.com/es/tutorial/scope-of-variables-python

39 40 41 Crear Comentarios en Python de la Forma Correcta - Kinsta®
https://kinsta.com/es/blog/comentarios-python/

42 5. El sistema de importación — Documentación de Python en Español -- 3.10.0
https://python-docs-es.readthedocs.io/es/3.10/reference/import.html