



Python: Conceptos de aplicaciones en producción

PYTHON: MÓDULOS Y
PAQUETES



Recordando

Un módulo sencillo

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

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

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



¿Qué es un módulo?

Si usamos el intérprete interactivo, todo lo que creamos se pierde el momento en que la sesión se cierra.

Un programa en general vivirá dentro de un archivo o script que podamos modificar o cambiar a nuestro gusto.

A medida que agreguemos complejidad, quizás empecemos a crear funciones que reutilicemos en diferentes sitios de nuestro programa.



¿Qué es un módulo?

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

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

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



¿Qué es un módulo?

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

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

`mi_modulo.py`

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

`otro_modulo.py`

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

O desde el intérprete interactivo

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



¿Qué es un módulo?

Alternativamente, podemos importar las funciones directamente

mi_modulo.py

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

otro_modulo.py

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

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

mi_modulo_1.py

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

mi_modulo_2.py

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

otro_modulo.py

```
# Incluimos los otros módulos en el espacio de nombres actual  
import mi_modulo_1  
import mi_modulo_2  
  
print(mi_modulo_1.mutiplicar_por_tres(5)) #15  
print(mi_modulo_2.mutiplicar_por_tres(5)) #15
```

Espacios de nombres (namespaces)



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

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

```
from mi_modulo_1 import multiplicar_por_tres
from mi_modulo_2 import multiplicar_por_tres

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




Espacios de nombres:

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

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

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

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

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



Librería estándar

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

Algunos módulos de fábrica

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

[Python Module Index – Python 3.12 documentation](https://docs.python.org/3.12/module-index.html)



Continuando

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

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

[Python Module Index — Python 3.12 documentation](#)



Importando

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

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

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



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





Importando

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

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

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





Recomendacion

Para evitar confusión y problemas con módulos y dependencias, usar los patrones recomendados por estilas de guía como PEP8 o por ejemplo Google Python style doc.

[styleguide | Style guides for Google-originated open-source projects](#)



Módulos como scripts

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

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

modulo_ejemplo.py

```
def mi_funcion(a, b):  
    print(a, b)  
  
print("Hola mi nombre es modulo_ejemplo")
```



Módulos como scripts

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

modulo_ejemplo.py

```
def mi_funcion(a, b):  
    print(a + b)  
  
print("Hola mi nombre es modulo_ejemplo")
```

Importado

otro_modulo.py

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

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

Ejecutado como script

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




Módulos como scripts

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

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

Sí, usemos la variable `__name__`!



__name__

Variable “mágica” disponible en cualquier módulo por defecto.

Python la define de dos maneras diferentes.

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

Es decir, tenemos mucha flexibilidad.



__name__

modulo_ejemplo.py

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

Ejecutado como script

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

Importado

otro_modulo.py

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

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



Python: Encuesta

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

utilidades.py

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

mi_programa.py

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

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

La encuesta se abrirá en Zoom



Python: Encuesta

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

utilidades.py

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

mi_programa.py

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

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

La encuesta se abrirá en Zoom



__name__

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

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

dir



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

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



Encontrando módulos

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

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

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

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

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



Encontrando módulos

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

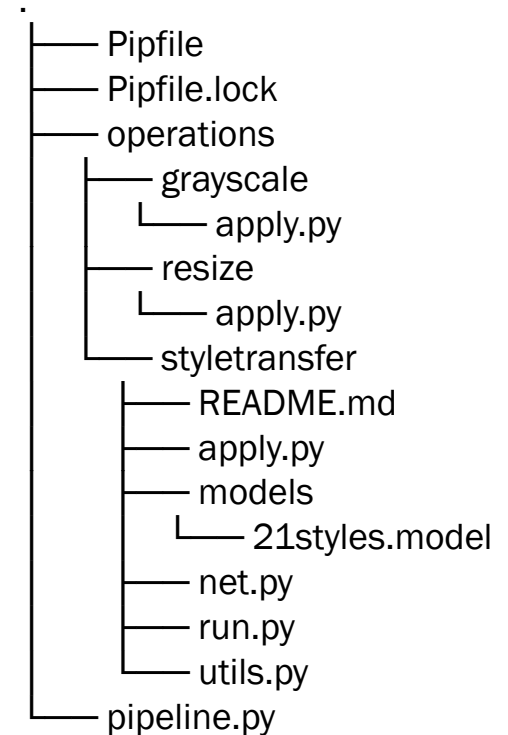
Podemos adicionar directorios cambiando la variable **sys.path**, que es simplemente una lista.



Estructura de carpeta

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

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





Paquetes

Una forma de agrupar los módulos de manera estructurada.

Un proyecto de Python que define un paquete simplemente requiere un árbol de carpetas. Esta jerarquía se accede en el código mediante puntos.

Por ejemplo para acceder al módulo **eco** uno puede usar

```
import sonido.efectos.eco
```

```
sonido/                               Paquete de nivel más alto
  __init__.py                         Script de inicialización de este paquete
  formatos/                           Subpaquete
    __init__.py
    mp3.py
    ...
  efectos/                             Subpaquete
    __init__.py
    eco.py
```



`__init__.py`

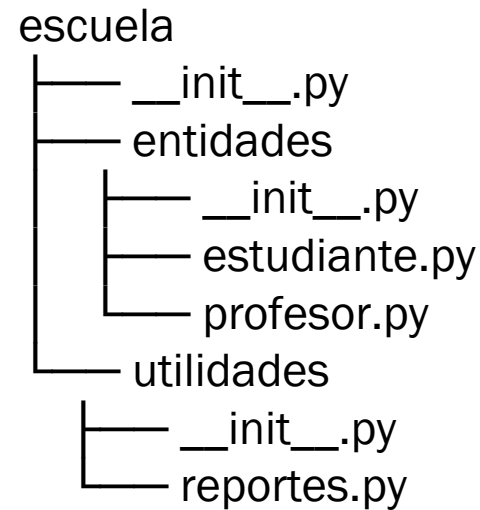
Un archivo obligatorio cuando se quiere tener un paquete. Esto permite a Python identificar un directorio como una jerarquía que debe tratarse como tal.

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



Python: Encuesta

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

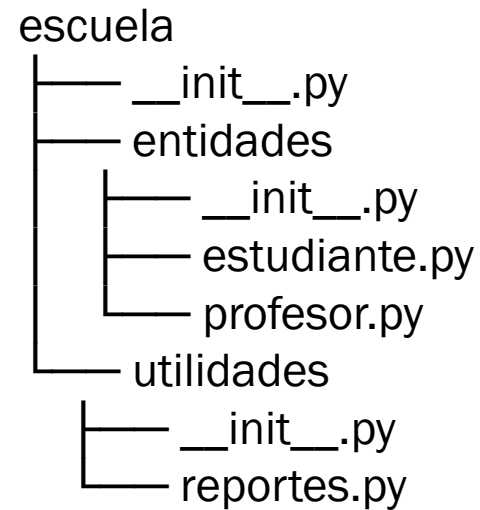


La encuesta se abrirá en Zoom



Python: Encuesta

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



Respuesta: `import escuela.entidades.estudiante`



Conclusión: Módulos y paquetes

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



Dependencias y paquetes externos

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

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



Dependencias en Python: pip

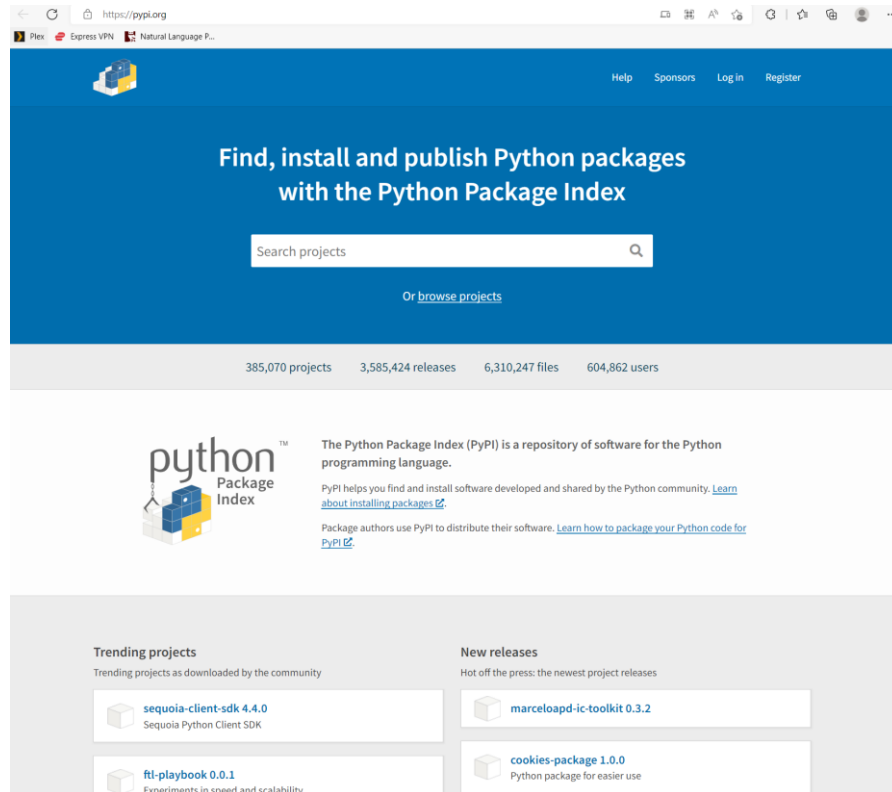
Al ser un proyecto popular de código abierto existe una amplia comunidad de soporte y contribuciones en muchos proyectos.

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

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



Dependencias en Python



El índice público de paquetes (PyPI)

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

[PyPI · The Python Package Index](https://pypi.org)



Dependencias en Python

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

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



Dependencias en Python

El comando inferior, instalará el paquete deseado desde el repositorio público de paquetes

```
python -m pip install AlgunPaquete
```

Adicionalmente, se puede especificar las versiones deseadas en caso de ser necesario.

```
python -m pip install AlgunPaquete==1.0.4
```

[Installing Python Modules — Python 3.12 documentation](https://docs.python.org/3.12/installing/index.html)



Dependencias en Python

El comando inferior, instalará el paquete deseado desde el repositorio público de paquetes

```
python -m pip install AlgunPaquete
```

Adicionalmente, se puede especificar las versiones deseadas en caso de ser necesario.

```
python -m pip install AlgunPaquete==1.0.4
```

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

[Installing Python Modules — Python 3.12 documentation](#)

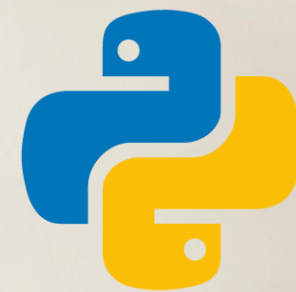


Dependencias en Python

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

```
Administrator: Command Prompt - py
C:\Users\deort>py -m pip install flask
Collecting flask
  Downloading Flask-2.1.2-py3-none-any.whl (95 kB)
----- 95.2/95.2 KB 1.8 MB/s eta 0:00:00
Collecting itsdangerous>=2.0
  Downloading itsdangerous-2.1.2-py3-none-any.whl (15 kB)
Collecting Werkzeug>=2.0
  Downloading Werkzeug-2.1.2-py3-none-any.whl (224 kB)
----- 224.9/224.9 KB 13.4 MB/s eta 0:00:00
Collecting Jinja2>=3.0
  Downloading Jinja2-3.1.2-py3-none-any.whl (133 kB)
----- 133.1/133.1 KB 8.2 MB/s eta 0:00:00
Collecting click>=8.0
  Downloading click-8.1.3-py3-none-any.whl (96 kB)
----- 96.6/96.6 KB ? eta 0:00:00
Collecting colorama
  Downloading colorama-0.4.5-py2.py3-none-any.whl (16 kB)
Collecting MarkupSafe>=2.0
  Downloading MarkupSafe-2.1.1-cp310-cp310-win_amd64.whl (17 kB)
Installing collected packages: Werkzeug, MarkupSafe, itsdangerous, colorama, Jinja2, click, flask
WARNING: The script flask.exe is installed in 'C:\Users\deort\AppData\Local\Programs\Python\Python310\Scripts' which is not on PATH.
Consider adding this directory to PATH or, if you prefer to suppress this warning, use --no-warn-script-location.
Successfully installed Jinja2-3.1.2 MarkupSafe-2.1.1 Werkzeug-2.1.2 click-8.1.3 colorama-0.4.5 flask-2.1.2 itsdangerous-2.1.2
WARNING: You are using pip version 22.0.4; however, version 22.1.2 is available.
You should consider upgrading via the 'C:\Users\deort\AppData\Local\Programs\Python\Python310\python.exe -m pip install --upgrade pip'
command.

C:\Users\deort>py
Python 3.10.5 (tags/v3.10.5:f377153, Jun 6 2022, 16:14:13) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import flask
>>> dir(flask)
['Blueprint', 'Config', 'Flask', 'Markup', 'Request', 'Response', '_builtins_', '_cached_', '_doc_', '_file_', '_loader_', '_name_', '_package_', '_path_', '_spec_', '_version_', '_app_ctx_stack', '_request_ctx_stack', '_abort', '_after_this_request', '_pp', '_appcontext_popped', '_appcontext_pushed', '_appcontext_tearing_down', '_before_render_template', '_blueprints', '_cli', '_config', '_copy_current_request_context', '_ctx', '_current_app', '_escape', '_flash', '_g', '_get_flashed_messages', '_get_template_attribute', '_globals', '_not_request_exception', '_has_app_context', '_has_request_context', '_helpers', '_json', '_jsonify', '_logging', '_make_response', '_message_flashed', '_redirect', '_render_template', '_render_template_string', '_request', '_request_finished', '_request_started', '_request_tearing_down', '_scaffold', '_send_file', '_send_from_directory', '_session', '_sessions', '_signals', '_signals_available', '_stream_with_context', '_templated_rendered', '_templating', '_typing', '_url_for', '_wrappers']
>>>
```



Python: Conceptos de aplicaciones en producción

PYTHON: ANOTACIONES DE
TIPO



Recordando tipos

Python es un lenguaje dinámico, es decir, las variables pueden acceder a cualquier tipo y no es necesario que el mismo se preserve durante la duración del script siendo ejecutado.

```
a = 1
```

```
a = "abcd"
```




Anotaciones de tipo

A pesar de que Python no verifica el tipado de las variables durante la ejecución del código, a veces es importante proveer anotaciones, o indicaciones que permitan al usuario entender las características del código siendo interpretado.

Asimismo, permite a herramientas automáticas detectar errores que el usuario puede haber ignorado.

Introducidas en Python 3.5



Anotaciones de tipo

Veamos como en C por ejemplo podemos ver que los tipos vienen por defecto, y de hecho no podemos evitarlos ya que son chequeados por el compilador al momento de transformar el código.

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

Python no los requiere, y de hecho permitirá que el código corra hasta el momento en que exista un problema (por ejemplo, aplicar una operación matemática a una cadena de texto).



Anotaciones de tipo

Hay ventajas para la falta de tipos ya que permite que el código sea más simple, y en algunos casos reduzca su complejidad.

Sin embargo, hay beneficios de tener tipos ya que permite que nuestro código sea explícito y no tenga ambigüedad. Es decir, podemos prevenir que una variable anotada como entero no se convierta en cadena de texto por accidente, o evitar pasar argumentos incorrectos a nuestras funciones.



Anotaciones de tipo

En Python las anotaciones de tipo no son como en lenguajes como C, en donde usar el tipo incorrecto terminará la ejecución o compilación del programa con errores.

¿Entonces para qué usarlas?

Porque como vimos antes, declarar tipos permite que nuestro código sea más explícito y claro.

Usándolas, herramientas conocidas como **linters** pueden detectar que el código no esté haciendo algo que no debe.



Anotaciones de tipo

- Mejora la legibilidad y claridad del código
- Permite a su entorno de desarrollo (Visual Studio, Pycharm, etc.) usarlas para proporcionar mejor funcionalidad de verificación durante el desarrollo.
- Mejor documentación del código durante el desarrollo de manera declarativa
- **¡Soporte de herramientas para automatizar chequeos estáticos!**



Anotaciones de tipo

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

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

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

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

Como podemos notar, mediante los dos puntos podemos determinar que clase de variable tenemos, en el caso de las funciones, podemos asimismo indicar el tipo retornado por la misma ->.



Anotaciones de tipo: Alerta

Usar anotaciones de tipo permite a nuestro IDE, *linter*, o entorno de desarrollo señalar problemas en nuestro código a manera de **alerta temprana**, antes de que el mismo llegue a las manos de sus usuarios.

Eventualmente aprenderemos como automatizar estos chequeos durante nuestro proceso de desarrollo.



Combinando tipos

Además de los tipos básicos `int`, `str`, `float`, `dict`, etc. También podemos combinarlos para crear definiciones de tipos más complejas. Por ejemplo, dos una variable que querremos sea de varios tipos.

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




Tipado en clases

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

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

Un mundo más completo

Podemos crear uniones, y muchas clases de tipos dinámicos, tipos condicionales.

Para aprender más, dirigirse a

[typing — Support for type hints — Python 3.12.0](#)