Python en Ámbitos Científicos

Facundo Batista & Manuel Carlevaro

27 de junio de 2024

VERSIÓN PRELIMINAR

Título: Python en Ámbitos Científicos
Autores: Facundo Batista & Manuel Carlevaro
ISBN-13 (versión electrónica): ???-?-????-????-?

Primera Edición (versión preliminar)
Escrito con XeLaTeX.

Licencia: Creative Commons Atribución-NoComercial-CompartirIgual 4.0 Internacional (CC BY-NC-SA 4.0)
Lugar: Olivos y La Plata, Buenos Aires, Argentina
Año: 2024
Web: http://pyciencia.taniquetil.com.ar/

10 9 8 7 6 5 4 3 2 1

VERSIÓN PRELIMINAR

Prefacio

Cuando la Comunidad Científica notó que las computadoras podían facilitar su trabajo, sus integrantes se convirtieron posiblemente en los adoptantes tempranos más entusiastas de la tecnología. Aunque el uso y la potencia de las computadoras crece continuamente en el ámbito científico, poco se ha avanzado en la formación y desarrollo de habilidades en Desarrollo de Software.

El cálculo científico requiere realizar combinaciones de múltiples tareas de diversa clase. Por ejemplo, es necesario registrar automáticamente datos de un experimento y visualizarlos, realizar cálculos numéricos o simbólicos, ordenar, clasificar, simular, etc. Muchas veces podemos utilizar paquetes de software que realizan esas tareas por nosotros, pero muchas otras ocurre que nadie ha implementado un determinado cómputo en la forma que necesitamos, o simplemente queremos probar ideas nuevas. Cualquiera sea la naturaleza de nuestra actividad en la ciencia o en la tecnología, no resulta infrecuente la necesidad de interactuar con computadoras a través de programas propios.

El uso de Python en aplicaciones científicas ha aumentado sostenidamente en los últimos años, sin embargo es difícil encontrar libros o manuales en castellano de Python que no estén pensados para programadores.

Este libro nace con la idea primaria de acercar Python al mundo científico, en un libro pensado para científiques, a partir de nuestra experiencia en el dictado del curso "Herramientas Computacionales para Científicos"que ofrecemos en la Universidad Nacional de La Plata y la Universidad Tecnológica Nacional, desde 2007. De la misma manera, la elección del castellano como idioma de escritura es un factor crítico, porque aunque sabemos que el inglés es una herramienta fundamental tanto para programar como para hacer ciencia, estamos convencidos que no debería ser una barrera de entrada.

Más allá de las secciones básicas de un libro (índice, bibliografía, etc.), el libro tiene tres grandes partes. La primera habla de Python, tanto de forma introductoria como también sobre otros temas que son fundamentales y algunas bibliotecas importantes. La segunda trata algunas herramientas fundamentales que son base para el trabajo en el resto del libro. Finalmente la tercera parte muestra cómo abordar temas científicos básicos utilizando Python, de forma teórica y práctica.

En todos los casos este libro esquiva la pretensión de ser una referencia absoluta, sino que tiene el propósito de allanar el camino de les científiques para dar los primeros pasos en el lenguaje y solucionar los problemas básicos (pero no por eso menos importantes) de la ciencia y la tecnología.

Olivos y La Plata, Buenos Aires, Argentina,

Facundo Batista & Manuel Carlevaro

Índice general

Parte I
Python

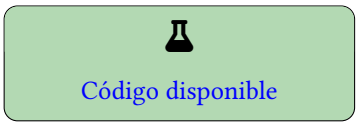
Esta primera parte comprende varios capítulos orientados a proveer la información necesaria de Python para poder entender el resto del libro.

Se sugiere a los lectores que no tienen experiencia previa en el uso de Python, un recorrido secuencial a través de los capítulos que componen esta Parte.

1 | Python más avanzado

Este capítulo está dedicado a temas breves avanzados de Python (o no tan introductorios) que son de utilidad en la programación en general y científica en particular.

Cada tema es independiente de la otro, entonces se puede evitar una lectura secuencial y consultar directamente la sección correspondiente al tema buscado.



1.1. Generadores

Llamamos "generador" a una forma especial de función que al ser ejecutada devuelve un objeto que cuando lo iteremos, irá "generando y devolviendo valores". Si la miramos es muy parecida a una función común excepto que contiene una o más expresiones `yield`.

El `yield` es el que cambia el juego. A diferencia de una función común que cuando la ejecutamos siempre arranca desde el principio y sigue hasta que se acaba el bloque de código o llega a un `return`, el generador ejecuta hasta que llega al `yield`, devolviendo el valor indicado, y cuando volvemos a pedirle un valor, al iterarlo, continúa "desde donde estaba".

Veamos esto en un ejemplo.

Cell 01

```
def supergen():
    print("dentro", 1)
    yield "a"
    print("dentro", 2)
    yield "b"
    print("dentro", 3)

for val in supergen():
    print("val", val)
```

```
dentro 1
val a
dentro 2
val b
dentro 3
```

Ejecutamos la función e iteramos lo que nos genera usando un `for`. En realidad acá no se llega a ver el efecto de "ir y volver" al generador, así que mejor mostrarlo si iteramos "a mano" usando la función integrada `next`.

El primer paso es ejecutar la función, que como tiene algún **`yield`** dentro directamente devuelve el generador pero sin avanzar en la ejecución del código (¡no muestra nada!):

CELL 02

```
gen = supergen()
gen
```

```
<generator object supergen at 0x7f252983c890>
```

Cuando le pedimos el primer valor, vemos que empieza a ejecutar el código, todas las lineas hasta que llega al primer **`yield`**, donde nos devuelve ese valor:

CELL 03

```
val = next(gen)
```

```
dentro 1
```

CELL 04

```
val
```

```
'a'
```

En este momento el generador está "suspendido" en el **`yield`**. Cuando le volvemos a pedir un valor **no** va a arrancar desde cero como cualquier función sino que va a continuar a partir del yield; en el caso de nuestro ejemplo efectivamente imprimiendo el `2` y luego devolviéndonos la letra `b`:

CELL 05

```
val = next(gen)
```

```
dentro 2
```

CELL 06

```
val
```

```
'b'
```

Si ahora le volvemos a pedir un valor va a continuar desde el segundo **`yield`**: imprimirá el `3` y luego generará una excepción en particular para denotar que se terminaron las posibles iteraciones, el `StopIteration`.

CELL 07

```
next(gen)
```

```
dentro 3

StopIteration                             Traceback (most recent call last)
Input In [7], in <cell line: 1>()
----> 1 next(gen)

StopIteration:
```

Tengamos en cuenta que esta excepción siempre sucede pero raramente la vemos; el `for`, por ejemplo, itera al generador hasta recibir esta excepción, y en ese momento lo único que hace es salir de su bucle.

La ventaja más directa de un generador es que si necesitamos una secuencia de valores, en lugar de construir una lista para guardar todos esos valores, nos irá devolviendo uno por uno.

Supongamos que queremos mostrarle al usuario algunos números de la secuencia de Fibonacci; necesitamos algo que nos genere esa secuencia, y desde afuera decidimos qué números mostrar y cómo. Usando una función normal nos quedaría:

CELL 08

```
def fibonacci(limit):  # función clásica
    numbers = [1]
    a, b = 0, 1
    while True:
        a, b = b, a + b
        if b > limit:
            break
        numbers.append(b)
    return numbers

for n in fibonacci(10):
    print(n)
```

```
1
1
2
3
5
8
```

En este ejemplo realmente no es necesario construir toda la lista para luego mostrarla ni tener todos los números al mismo tiempo, con tener los valores al momento de mostrarlos alcanza. Esto no sólo mejora la utilización de memoria de nuestro programa sino también la performance, ya que evitamos pedirle memoria al sistema operativo una y otra vez para ir creciendo la lista.

CELL 09

```
def fibonacci(limit):  # generador
    yield 1
    a, b = 0, 1
    while True:
        a, b = b, a + b
        if b > limit:
            break
        yield b

for n in fibonacci(10):
    print(n)
```

```
1
1
2
3
5
8
```

Otra mejora con respecto a las funciones es que podemos tener generadores "infinitos" lo cual es útil cuando a priori no tenemos un límite. Adaptando el ejemplo anterior, tendríamos:

CELL 10

```
def fibonacci():  # generador infinito
    yield 1
    a, b = 0, 1
    while True:
        a, b = b, a + b
        yield b

for n in fibonacci():
    if n > 10:
        break
    print(n)
```

```
1
1
2
3
5
8
```

Aquí vemos como `fibonacci` nos irá dando números para siempre, y es desde afuera que controlamos cuando cortar y por qué; en el ejemplo es luego de pasar un límite, pero podría ser cualquier situación (el usuario dejó de hacer click, etc.) y no tenemos que incluir esa lógica en la función generadora en sí.

Aunque el **`yield`** convierta a la función en un generador también podemos tener **`return`** en su bloque de código. En el contexto de un generador, el **`return`** generará una excepción `StopIteration`, como si hubiese terminado. Veamos el siguiente ejemplo:

Cell 11

```
def foo(simple):
    yield 1
    if simple:
        return
    yield 2
```

Cell 12

```
list(foo(simple=True))
```

```
[1]
```

Cell 13

```
list(foo(simple=False))
```

```
[1, 2]
```

Si el **return** especifica un objeto, este será usado para instanciar `StopIteration` (tengamos en cuenta que muchas veces no se muestra esta excepción, como cuando usamos el **for**, así que el objeto especificado no siempre será útil).

Los generadores tienen una funcionalidad que no es tan conocida: podemos influir desde afuera sobre sus estados. Por ejemplo podemos pasarle valores al **yield**, como en el siguiente generador que primero entrega un 5 y al ser continuado recibe un valor, que lo muestra y lo usa para devolverlo multiplicado por 3:

Cell 14

```
def foo():
    a = yield 5
    print("en el generador:", a)
    yield a * 3
```

Entonces creamos el generador y lo avanzamos hasta el primer **yield**, nos devuelve el valor esperado, lo mostramos. El próximo paso es usando un método nuevo, el `send`, para *enviarle* un valor al generador; este valor será mostrado por el generador y nos devolverá su múltiplo:

Cell 15

```
g = foo()
val = next(g)
print("afuera:", val)
```

```
afuera: 5
```

Cell 16

```
val = g.send(3)
print("afuera:", val)
```

```
en el generador: 3
afuera: 9
```

También podemos indicarle al generador que genere una excepción (con el método `throw`) o

directamente cerrarlo (con el método `close`) de manera que su próxima iteración genere `StopIteration`.

Todas estas capacidades juntas de los generadores, no sólo la de quedar suspendidos y luego reaunudarse con el estado anterior, sino también la de poderse enviar "mensajes" entre ellos hacen que los generadores sean los predecesores naturales de las "corrutinas" en el mundo asincrónico de Python, del que hablamos en **??**.

1.2. Administradores de contexto

Una de las formas más comunes de encapsular código son las funciones. Nos permiten proveer una determinada funcionalidad que será usada en distintas partes de un programa. Una de las limitaciones de las funciones es que proveen un sólo punto de entrada, entonces se quedan cortas cuando la funcionalidad que queremos proveer debe ejecutarse en parte *antes* de un determinado código, y también *después* de ese código.

Veamos eso en un pequeño pseudocódigo. Supongamos que necesitamos preparar un recurso antes de la ejecución de determinado código, y luego de esa misma ejecución cerrarlo o darlo por terminado; tendríamos algo como lo siguiente:

```
(... debemos preparar el recurso ...)
arranca el código que usa el recurso
...
termina el código que usa el recurso
(... debemos terminar o cerrar ese recurso ...)
```

Ya dijimos que no podemos utilizar una función para toda esa funcionalidad, ¡pero podemos usar dos! Una que prepare el recurso, y otra que lo termine.

```
resource_setup()
arranca el código que usa el recurso
...
termina el código que usa el recurso
resource_ending()
```

Incluso la primer función podría devolver el recurso a utilizar. Y para asegurar que el recurso se finalice apropiadamente siempre deberíamos soportar el caso en que el código que usa el recurso genere una excepción. Nos va quedando algo como lo siguiente:

```
resource = resource_setup()
try:
    # arranca el código que usa el recurso
    ...
    # termina el código que usa el recurso
finally:
    resource_ending()
```

Los administradores de contexto vienen a formalizar toda esta idea y al mismo tiempo simplificar nuestro código cuando los usamos. Encapsula las dos funciones que necesitamos (de pre-

paración y de terminación) dentro de un objeto, y marca de forma explícita el bloque de código que usa el recurso (con la sangría luego de usar el **with**):

```
with resource_manager() as resource:
    # arranca el código que usa el recurso
    ...
    # termina el código que usa el recurso
```

Ese `resource_manager` que ejecutamos en nuestro código ejemplo es el que devuelve una instancia del administrador de contexto. Este administrador de contexto tiene dos métodos, uno que se ejecuta antes de entrar al bloque de código, y otro que se ejecuta al salir del bloque de código (más allá de que se terminó porque se ejecutaron todas las líneas u ocurrió una excepción). Estos dos métodos son los equivalentes a las dos funciones que usábamos arriba.

Veamos dos ejemplos reales. En el primer caso usamos la función integrada `open` que devuelve un objeto que funciona como administrador de contexto, y nos asegura que el archivo va a estar abierto correctamente antes de arrancar el bloque de código, y se va a cerrar cuando este termine.

```
with open("/tmp/prueba.txt") as fh:
    print(fh.read())
```

Notemos que en este caso de uso el administrador de contexto nos devuelve como recurso a utilizar el mismo *file handler* que usamos para leer del archivo. Es distinto al siguiente caso real donde tomamos un lock antes de entrar al bloque de código y lo liberamos luego, pero donde no necesitamos hacer referencia al recurso desde el bloque de código manejado:

```
with threading.Lock():
    do_something()
    # etc
```

Para construir nuestro propio administrador de contextos sólo tenemos que entender cómo Python va a llamar a las dos funciones que ya sabemos que necesitamos proveer, la de entrada al bloque de código para preparar el recurso, y la de salida para ocuparnos del mismo. En realidad esas dos funciones son métodos del objeto que proveemos, y tienen que tener un nombre fijo.

La primera es `__enter__`, no recibe nada, y lo que potencialmente devuelva se vincula al objetivo del **with** (el que indicamos con el **as**). La segunda es `__exit__` y recibe tres parámetros y tiene un comportamiento especial con su resultado, dependiendo si en el bloque de código sucedió una excepción o no. En caso de tener una excepción en el bloque de código recibirá el tipo y valor de la misma, y el *traceback* correspondiente, y si la función devuelve True esa excepción será suprimida (en caso contrario seguirá su curso natural). Si el bloque de código se completó correctamente, los parámetros de la función estarán en None y no importa para nada lo que devuelva.

A modo de ejemplo armemos un administrador de contexto que nos provea un directorio temporal para trabajar dentro del mismo.

CELL 01

```
import os
import shutil


class TempDir:

  def __init__(self, dirname):
    self.dirname = dirname

  def __enter__(self):
    os.makedirs(self.dirname, exist_ok=True)

  def __exit__(self, exc_type, exc_value, tb):
    shutil.rmtree(self.dirname)
```

Tenemos una clase `TempDir` que al instanciarla recibe el nombre o *path* del directorio que se hará temporal, el cual se guarda en un atributo interno. Además de `__init__` la clase provee los dos métodos necesarios para funcionar como un administrador de contexto. En el `__enter__` crea todos los directorios que correspondan (sin fallar si ya existen), y en el `__exit__` borra ese directorio (sin importar lo que haya dentro). Notemos también que aunque `__exit__` recibe los tres parámetros correspondientes a una posible excepción, no hace nada con ellos: no importa si hubo una excepción o no, el directorio se borra.

CELL 02

```
import os

example_dir = "/tmp/testbook"
print("¿Existe previamente?", os.path.exists(example_dir))
```

```
¿Existe previamente? False
```

CELL 03

```
with TempDir(example_dir):
    print("¿Existe durante la ejecución interna?", os.path.exists(example_dir))
    testfile = os.path.join(example_dir, "prueba.txt")
    with open(testfile, "wt") as fh:
        fh.write("prueba")
    print("¿Pudimos usar ese directorio?", os.path.exists(testfile))

print("¿Existe al salir del with?", os.path.exists(example_dir))
```

```
¿Existe durante la ejecución interna? True
¿Pudimos usar ese directorio? True
¿Existe al salir del with? False
```

Antes de usarlo verificamos que el directorio de prueba no existe, y luego para usarlo instanciamos `TempDir` con ese directorio. En el bloque de código del **`with`** vemos que el directorio sí existe, e incluso creamos un archivo dentro. Finalmente, luego de salir del bloque de código del **`with`** (como denota el sangrado) vemos que el directorio no existe más.

Hay otra forma de construir administradores de contexto que a priori parece más simple, sacrificando un poco de flexibilidad. Esta segunda forma es escribiendo un generador que, al

decorarlo de forma particular, nos va a proveer el comportamiento deseado.

La sintaxis es la siguiente:

```
from contextlib import contextmanager

@contextmanager
def nuestro_contextmanager(...):
    resource = resource_setup()
    try:
        yield resource
    finally:
        resource_ending()
```

Notemos como esta estructura de código es muy parecida a la que teníamos cuando empezamos a explicar administradores de contexto al principio de la sección. Obviamente, no hace falta llamar a las funciones de preparado y terminación del recurso, ese código podría estar directamente en nuestro generador. El truco de esta sintaxis y el decorador `contextmanager` es que nuestro generador va a proveer el recurso (que se vinculará con el **`as`** del **`with`**) en el único **`yield`** que debe tener.

Luego dependerá de nosotros tener una estructura try/finally o try/except/finally alrededor del **`yield`** para reaccionar o no a posibles excepciones del bloque de código del **`with`**.

Escribamos nuevamente nuestro administrador de contexto de ejemplo pero utilizando esta forma:

Cell 04

```
from contextlib import contextmanager

@contextmanager
def temp_dir_v2(dirname):
    os.makedirs(dirname, exist_ok=True)
    try:
        yield
    finally:
        shutil.rmtree(dirname)
```

Cell 05

```
print("¿Existe previamente?", os.path.exists(example_dir))

with temp_dir_v2 (example_dir):
    print("¿Existe durante la ejecución interna?", os.path.exists(example_dir))
    testfile = os.path.join(example_dir, "prueba.txt")
    with open(testfile, "wt") as fh:
        fh.write("prueba")
    print("¿Pudimos usar ese directorio?", os.path.exists(testfile))

print("¿Existe al salir del with?", os.path.exists(example_dir))
```

```
¿Existe previamente? False
¿Existe durante la ejecución interna? True
¿Pudimos usar ese directorio? True
¿Existe al salir del with? False
```

Quedará en la evaluación de cada caso si es lo mejor usar una forma o la otra para escribir un administrador de contexto, en función de si queremos priorizar la flexibilidad o tener un código más conciso.

1.3. Acercándonos a la programación funcional

Sabemos que Python es un lenguaje que permite una multiplicidad de estilos de programación. Podemos hacer un script puramente declarativo o sistemas cien por ciento basados en la programación orientada a objetos. Y también nos provee herramientas que nos acerca a la programación funcional.

La programación funcional descompone un problema en un conjunto de funciones. Idealmente, las funciones solo reciben entradas y producen salidas, y no tienen ningún estado interno que afecte la salida producida para una entrada dada.

En esta sección estudiaremos cuatro detalles que acercan Python a lo que esperaría encontrar alguien que se acerca al lenguaje desde la programación funcional: la habilidad de definir funciones dentro de una expresión (que en Python se llaman "funciones lambda") y las tres operaciones básicas *map*, *filter* y *reduce*.

1.3.1. Funciones lambda

Una función lambda es una función como las que normalmente definimos con `def` pero son anónimas (no tienen un nombre) y su sintaxis hace que permitan ser definidas dentro de otra expresión.

Por ejemplo, supongamos que queremos ordenar una lista de palabras poniendo primero aquellas cuya longitud está más cercana a 10. Para ello usamos la función integrada `sorted` que acepta una parámetro `key` que justamente recibe una función que devuelve el valor que se toma en cuenta para realizar el orden. Escribamos esto usando una función "normal":

Cell 01

```
def ordenadora(palabra):
    """Devuelve la diferencia entre el largo de la palabra y 10."""
    return abs(10 - len(palabra))
```

Cell 02

```
palabras = ["camión", "mono", "heladera", "televisión", "pared", "banana"]
sorted(palabras, key=ordenadora)
```

```
['televisión', 'heladera', 'camión', 'banana', 'pared', 'mono']
```

La función es tan sencilla y tan orientada a ser usada sólo dentro del `sorted` que estamos ejecutando, que es una buena oportunidad para convertirla en *lambda*:

Cell 03

```
sorted(palabras, key=lambda palabra: abs(10 - len(palabra)))
```

```
['televisión', 'heladera', 'camión', 'banana', 'pared', 'mono']
```

En esta última linea vemos que a `key` le pasamos directamente la función que necesitamos, que recibe `palabra` y devuelve la distancia calculada.

La sintaxis de la función lambda es

```
"lambda" [lista_de_parámetros] ":" expresión
```

La lista de parámetros es similar a las funciones definidas con **`def`**, pero la función en vez de tener todo un bloque de código acepta solamente una expresión; el resultado de calcular esa expresión será lo que devuelve la función.

Cabe acotar que las funciones lambda no son particularmente especiales en Python: no consumen menos memoria ni son más rápidas. Si su sintaxis nos termina limitando para escribir una función que necesita ser más compleja (ya que las lambda no pueden tener declaraciones ni anotaciones), pasamos a utilizar una función clásica y listo.

1.3.2. map, filter y reduce

Estas tres funciones, tan conocidas en la programación funcional, operan básicamente aplicando una función a los elementos de un iterable.

Arranquemos explicando `map`. Su funcionamiento es simple: aplica una función a los valores de un iterable, generando cada uno de los resultados.

Veamos un ejemplo donde calculamos el largo de cada palabra usando la función integrada `len`:

Cell 04

```
palabras
```

```
['camión', 'mono', 'heladera', 'televisión', 'pared', 'banana']
```

Cell 05

```
map(len, palabras)
```

```
<map at 0x7feccdaf75b0>
```

El resultado, quizás no esperado, es un objeto `map` que va a generar los resultados cuando lo iteremos. Podemos hacer eso con un **`for`**:

Cell 06

```
for largo in map(len, palabras):
    print(largo)
```

```
6
4
8
10
5
6
```

En realidad para mostrar ejemplos o probar porciones de código muchas veces utilizamos `list` para consumir fácilmente el generador y tener una lista con los resultados (cosa que casi nunca hacemos en códigos reales, donde lo iteramos directamente al usarlo):

CELL 07

```
list(map(len, palabras))
```

```
[6, 4, 8, 10, 5, 6]
```

En ese mínimo ejemplo usamos la función integrada `len`, con lo cual el resultado son los largos de las palabras. Pero supongamos que queremos los cuadrados de algunos números. Necesitamos una función que no tenemos creada con anterioridad. La necesitamos definir nosotros, pero es tan sencilla que podemos usar una función lambda:

CELL 08

```
nros = [6, 4, 8, 10, 5, 6]
list(map(lambda n: n ** 2, nros))
```

```
[36, 16, 64, 100, 25, 36]
```

En realidad no hace falta que sean funciones lambda, puede ser cualquier función, pero la posibilidad de definir la función en la misma linea es muy cómodo en algunos casos.

La función `filter` también recibe como parámetros una función y un iterable. Aplicará esa función a cada elemento del iterable, e irá devolviendo cada elemento solamente si el resultado de esa función es interpretable como verdadero. Dos detalles para resaltar: no devuelve el resultado de la función (como hacía `map`) sino que usa ese resultado para saber si tiene que devolver o no el elemento del iterable, y por otro lado no hace falta que la función devuelva True o False, sino que interpreta el valor booleano de ese resultado.

CELL 09

```
palabras
```

```
['camión', 'mono', 'heladera', 'televisión', 'pared', 'banana']
```

CELL 10

```
from collections import Counter

def muchas_a(palabra):
    return Counter(palabra)["a"] >= 2

list(filter(muchas_a, palabras))
```

```
['heladera', 'banana']
```

Como caso especial `filter` acepta **`None`** en vez de la función, cambiando levemente su comportamiento: en este caso devolverá los elementos que ellos mismos evalúen a verdadero. Veamos un ejemplo típico donde luego de procesar una bloque de texto queremos eliminar las líneas vacías:

Cell 11

```
texto = """
Arranca el texto.

Parte media.

Vamos cerrando.
"""
lineas = texto.split("\n")
lineas
```

```
['', 'Arranca el texto.', '', 'Parte media.', '', 'Vamos cerrando.', '']
```

Cell 12

```
list(filter(None, lineas))
```

```
['Arranca el texto.', 'Parte media.', 'Vamos cerrando.']
```

La función `reduce`, nuevamente, también recibe como parámetros una función y un iterable, pero a diferencia de las anteriores no aplica la función a cada elemento del iterable. Es más, la función no recibe un argumento sino dos. Al comenzar `reduce` llama a la función con el primero y el segundo elemento del iterable, y recibe su resultado; a partir de este momento irá llamando a la función con el resultado de la ejecución anterior y el próximo elemento del iterable.

Para visualizar mejor este proceso armemos una función que no sólo multiplica dos números sino que también previamente los muestra.

Cell 13

```
from functools import reduce

def multip(a, b):
    print("nros:", a, b)
    return a * b

result = reduce(multip, [1, 2, 3, 4, 5])
print("result:", result)
```

```
nros: 1 2
nros: 2 3
nros: 6 4
nros: 24 5
result: 120
```

Vemos en el listado que la función se ejecutó con los números 1 y 2 (de la lista fuente), luego con 2 (el resultado de la multiplicación de los anteriores) y 3 (de la lista fuente), luego con 6 (el resultado de la multiplicación de los anteriores) y 4 (de la lista fuente), etc.

Por supuesto, en estos casos también es muy práctico usar **`lambda`**:

Cell 14

```
reduce(lambda x, y: x * y, [1, 2, 3, 4, 5])
```

```
120
```

Notemos como a esta función en particular la tuvimos que importar del módulo `functools`: en el pasado lejano era una función integrada igual que `map` y `filter`, pero en la evolución a Python 3 se decidió meterla en el módulo porque no es tan usada.

1.3.3. Alejándonos de la programación funcional

Python agregó las funcionalidades que recién describimos en las primeras versiones del lenguaje, pero luego fueron siendo reemplazadas por otras construcciones más pytónicas.

En concreto, las comprensiones de listas (de las que hablamos en detalle en **??**) se solapan en funcionalidad con el `map` y `filter`, veamos algunos ejemplos usándolas por separado y en combinación.

El primer caso reemplaza el `map` por una *list comprehension* de forma bastante directa. En el caso del `map` convertimos el resultado en lista para poder mostrarlo, no hace falta hacer eso con la comprensión de lista porque justamente devuelve ese tipo de dato; por otro lado, en un código real para lograr la eficiencia del `map` (que devuelve un generador) deberíamos convertir la *list comprehension* a *generator comprehension*.

Cell 15

```
import math

# el logaritmo de unos valores
valores = [15, 0.32, 1.01, 17]
print(list(map(math.log, valores)))
print([math.log(x) for x in valores])
```

```
[2.70805020110221, -1.1394342831883648, 0.009950330853168092, 2.833213344056216]
[2.70805020110221, -1.1394342831883648, 0.009950330853168092, 2.833213344056216]
```

El segundo caso donde filtramos los resultados sí cambia un poco, porque con la comprensión de lista no hace falta crear una función que va a ser llamada para cada elemento, sino que eso es parte de la sintaxis. Esto hace al código mucho más eficiente, porque en Python las llamadas a función, por distintas características dinámicas del lenguaje, son relativamente caras.

Cell 16

```
# filtramos y nos quedamos con los valores mayores a 1
print(list(filter(lambda x: x > 1, valores)))
print([x for x in valores if x > 1])
```

```
[15, 1.01, 17]
[15, 1.01, 17]
```

Finalmente el ejemplo donde aplicamos tanto el logaritmo como el filtro:

Cell 17

```
# combinadas! el logaritmo de los valores mayores a 1
print(list(map(math.log, filter(lambda x: x > 1, valores))))
print([math.log(x) for x in valores if x > 1])
```

```
[2.70805020110221, 0.009950330853168092, 2.833213344056216]
[2.70805020110221, 0.009950330853168092, 2.833213344056216]
```

Aquí es más claro la mayor legibilidad de las comprensiones de listas, porque no sólo no obliga a crear nuevas funciones, sino que tampoco sufre del inconveniente del "anidado" (habiendo dicho eso recordemos que también podemos tener comprensiones de listas ilegibles, anidando unas con otras, pero en general en ese caso se termina escribiendo un `for` "clásico").

1.4. Pruebas de unidad

Las pruebas de unidad, en inglés *unit tests* son la forma más efectiva de verificar que un programa funciona correctamente.

Son pruebas "de unidad" porque la idea es que cada prueba valide la mínima unidad posible de código, tanto para simplificar la prueba en sí como para identificar mejor la falla encontrada si el test no termina satisfactoriamente. El otro extremo de este concepto son las pruebas "de integración" donde se prueban que las distintas partes del sistema (o los distintos sistemas) interactúen correctamente. En la práctica las pruebas de unidad no terminan siendo cien por ciento puras (sólo de unidad) y algún componente de integración tienen, lo cual es útil para simplificar las pruebas en sí.

El conjunto de pruebas de unidad de un determinado programa se ejecutan normalmente al momento de desarrollo para validar que los cambios hechos al agregar una funcionalidad o corregir un bug no van en detrimento de otras partes de ese programa. Tener un buen conjunto de pruebas de unidad permite lograr una mejora continua de la calidad del sistema; por supuesto siempre existe la posibilidad de que al modificar un programa introduzcamos un bug en alguna sección del código mal probada o directamente sin pruebas, pero si tenemos la política de ir agregando pruebas para todos esos casos la evolución favorable del programa estará asegurada, especialmente permitiendo el aumento de su complejidad sin que se vuelva inmanejable para el equipo de desarrollo.

También se acostumbra correr estas pruebas en sistemas automáticos antes de integrar el código propuesto a la rama principal, antes de empaquetar y distribuir el código, antes de poner una nueva versión del sistema a trabajar en un servidor, o en general antes de cualquier acción que potencialmente impacte en la usabilidad del programa (estos sistemas que corren las pruebas y luego ejecutan alguna acción determinada se llaman "CI/CD" en su conjunto, por *Continuous Integration / Continuous Delivery* (integración continua / entrega continua).

Un tema importante sobre el que hay muchas discusiones y puntos de vista es en qué momento hay que escribir el test correspondiente a un código. Un extremo es escribir primero todo el código en sí y luego todas las pruebas de unidad. Este caso nunca es recomendable, pero por supuesto si llegamos a un programa o sistema que ya está escrito y no tiene ninguna prueba, obviamente debemos empezar a construir el conjunto de pruebas de unidad en ese punto.

El otro extremo es lo que se llama *Test Driven Development* (TDD), "desarrollo dirigido por las pruebas", que versa que no debería escribirse código alguno sin antes haber hecho el test correspondiente.

Como todo en la vida, la realidad se desarrolla en los grises intermedios. En códigos grandes y más o menos estables recomendamos realizar la prueba o pruebas correspondientes antes de modificar el código para agregar funcionalidad o solucionar un problema. Pero en programas nuevos o scripts pequeños, o incluso en sistemas estables donde estamos explorando un cambio grande, siempre es recomendable avanzar con "código de exploración" hasta tener más o menos definida la estructura general del cambio o del programa nuevo y recién ahí agregar los tests para

ese código, momento a partir del cual cambiamos al modo más cercano a TDD para seguir con la evolución del sistema.

A nivel estructura de proyecto vamos a necesitar diferenciar dos aspectos de "tener pruebas de unidad". El primero es tener los archivos donde están escritos los diferentes tests, el segundo es definir y configurar qué herramienta vamos a utilizar para encontrar esos archivos con las pruebas, correrlos, y presentarnos un reporte con los resultados.

Hay algunas reglas para seguir a la hora de armar los archivos de prueba. Algunas son convenciones para simplificar el uso de los mismos, otros son defaults de la herramienta que corre pruebas y son más o menos configurables. Es por esto que debemos tomar las siguientes recomendaciones no como reglas absolutas sino como justamente lo que son, recomendaciones, y siempre habrán casos donde no se sigan al pie de la letra…

- ¿Dónde ponemos los archivos con las pruebas? En un directorio separado en la raíz de nuestro proyecto, para que sea fácil excluirlos a la hora de empaquetar nuestro proyecto para distribuirlo. Los archivos deben comenzar su nombre con `test_`.
- ¿Cuántos archivos de pruebas? Uno por módulo o script de nuestro proyecto, ya que ese paralelismo nos permite entender fácilmente dónde ir a buscar los tests correspondientes al código que vamos a tocar. Si nuestro proyectos tiene módulos separados en varios paquetes (directorios), imitaremos esa estructura dentro del directorio de tests.
- ¿Qué estructura interna en cada archivo? Acá hay dos mundos, influidos por las capacidades de las distintas herramientas para correr las pruebas. La herramienta que viene en la biblioteca estándar de Python, `unittest`, necesita que armemos clases que hereden de `unittest.TestCase` con métodos cuyos nombres arranquen con `test_`. El corredor de tests muy popular `pytest` (que es el que usaremos aquí en los ejemplos), aunque soporta la misma estructura de `unittest`, nos permite escribir directamente cada prueba como una función en el archivo (también arrancando sus nombres con `test_`).

Aunque hay algunos casos especiales más o menos soportados por las distintas herramientas, podemos asumir que cada prueba de unidad va a terminar en uno de los siguiente cuatro estados:

- OK: terminó bien, el código a probar se ejercitó correctamente y todas las verificaciones posteriores fueron exitosas.
- Falla (*failure*): nos indica que, aunque se ejercitó correctamente el código, alguna de las verificaciones que realizamos no fue exitosa
- Error: hubo algún problema no esperado al preparar la prueba o en las posteriores verificaciones
- Saltado (*skipped*): el test no se ejecutó por alguna condición especificada en el test mismo (por ejemplo, el test funciona sólo en un entorno Windows y el desarrollador ejecutó los tests en Linux)

Vayamos construyendo un ejemplo práctico para mostrar las distintas partes de un sistema con pruebas de unidad. Por lo mínimo del ejemplo no hace falta una estructura compleja de directorios, así que para simplificar el código en el repositorio del libro tendremos el módulo

con nuestra función y el archivo con las pruebas en el mismo directorio, e iremos cambiando de directorio con cada versión de código.

Supongamos entonces que tenemos el requerimiento de escribir una función que suma dos números. El requerimiento es tan simple que podemos arrancar directamente con las pruebas. Hacemos una básica, y sólo para no tener una sola, lo mismo pero al revés:

```
from mod import adder  # importamos la función a testear


def test_simple():
    result = adder(3, 5)
    assert result == 8


def test_reverse():
    result = adder(5, 3)
    assert result == 8
```

La estructura general de cada prueba se divide en tres partes: preparación de la prueba, ejecución del código a probar, verificaciones posteriores. La preparación de la prueba varía mucho según el caso, puede ser inexistente como en las dos pruebas que mostramos recién o de muchas lineas si el contexto necesario para probar el código es complejo. Luego se ejecuta el código a probar, lo cual casi siempre es llamar a una función y obtener un resultado de la misma (como en nuestros ejemplos recién mostrados) o esperar que se genere una excepción durante esa ejecución (más adelante veremos un ejemplo de esto). Y finalmente las verificaciones, que también pueden ser mínimas (como en nuestro caso) o complejas y en varios pasos.

Es tiempo de escribir nuestra función. Es muy simple:

```
def adder(n1, n2):
    return n1 + n2
```

Para correr estas pruebas elegimos `pytest` como herramienta. Usamos `fades` (como vimos en ??) para que lo instale en un entorno virtual y lo ejecutamos directamente desde allí (con el parámetro `-q` para que no nos dé un montón de información que en algunos casos es útil pero nos molestaría aquí en el libro; les dejamos como tarea probarlo sin esa opción o incluso con `-v` para que sea más verborrágico).

```
unittesting/v01 $ fades -d pytest -x pytest -q test_adder.py
..                                                                   [100%]
2 passed in 0.00s
```

Dos pruebas pasadas OK.

Nos damos cuenta que podríamos estar manejando más casos en nuestras pruebas: negativos, cero, etc. Más allá que a priori nos parece que nuestra función se comportaría correctamente con estos casos, no es mala idea probar extremos o condiciones menos comunes. Por supuesto, no podemos probar con *todos* los números posibles, pero siempre es recomendable pensar con cuales casos el código podría llegar a tener algún inconveniente.

Por otro lado, si vamos a hacer una función para cada uno de esos casos quedaría todo muy repetitivo. El módulo `pytest` nos resuelve esta situación proveyendo un decorador que parametriza el test. Entonces ahora nuestra función valida varios casos, recibiendo los números a sumar y la respuesta esperada; en realidad `pytest` termina ejecutando muchos tests, lo cual nos da la ventaja de la aislación de cada uno y tener como resultado una falla puntual para el caso indicado.

```
import pytest

from mod import adder


@pytest.mark.parametrize("a, b, c", [
    (3, 5, 8),
    (5, 3, 8),
    (0, 1000, 1000),
    (-23, 0, -23),
    (87, -87, 0),
    (-15, -15, -30),
])
def test_numbers(a, b, c):
    result = adder(a, b)
    assert result == c
```

El decorador de parametrización recibe dos valores: primero los nombres de los argumentos donde la función recibirá los valores, y luego una lista de los casos (en cada caso tenemos que tener la misma cantidad de valores que la cantidad de argumntos definidos). Tenemos total libertad con los nombres de los argumentos o su significado, en este caso usamos `a` y `b` para los dos números fuente y `c` para el resultado esperado.

Corremos las pruebas, vemos que efectivamente son varias:

```
unittesting/v02 $ fades -d pytest -x pytest -q test_adder.py
......                                                   [100%]
6 passed in 0.01s
```

Como siempre pasa en la vida real, eventualmente nos cambian las especificaciones de la tarea dada. Ahora la función tiene que soportar también que los números vengan como cadena. Por ejemplo, soportar el `"3"` como si fuese un `3`.

Como nuestra función ya está lo suficientemente estable, este cambio lo encaramos como recomendamos arriba: haciendo primero las pruebas. Agregamos entonces algunos tests a lo que ya teníamos de antes:

```
@pytest.mark.parametrize("a, b, c", [
    ("3", 5, 8),
    (87, "-87", 0),
])
def test_with_letters_ok(a, b, c):
    result = adder(a, b)
    assert result == c


```

```
@pytest.mark.parametrize("a, b", [
    ("", 5),  # empty
    (87, "foo"),  # not a number
])
def test_with_letters_bad_value(a, b, c):
    with pytest.raises(ValueError):
        adder(a, b)
```

En este caso no sólo agregamos los casos de éxito donde la suma que tenemos que verificar es la correcta, sino también casos donde el comportamiento esperado es que la función genere una excepción del tipo `ValueError` (cuando la cadena no representa de forma válida un número).

Al correr las pruebas nos encontramos con que tenemos una salida extensa. Primero el reporte de un caracter por prueba, luego la situación en detalle para cada prueba, y al final un resumen corto:

```
unittesting/v03 $ fades -d pytest -x pytest -q test_adder.py
......FFEE                                                          [100%]
================ ERRORS ================
(...)
=============== FAILURES ===============
(...)
======= short test summary info ========
(...)
2 failed, 6 passed, 2 errors in 0.02s
```

La última linea nos pone en palabras lo que nos iba informando `pytest` al correr prueba a prueba al principio: 6 casos estuvieron bien, tenemos dos fallas, y dos errores. Las fallas son esperables, porque agregamos pruebas pero todavía no adaptamos el código correspondientemente, pero los errores no deberían estar allí. Revisamos el mensaje (que omitimos en la salida de recien por brevedad) y encontramos:

```
  @pytest.mark.parametrize("a, b", [
      ("", 5),  # empty
      (87, "foo"),  # not a number
  ])
  def test_with_letters_bad_value(a, b, c):
E       fixture 'c' not found
```

Esto es que definimos una parametrización para la función usando dos argumentos, pero luego en la definición de la función nos quedó que aceptaba un tercero (el error viene de copiar la función de arriba donde el tercer valor era el resultado esperado, que en este caso no corresponde). Corregimos entonces el bug en nuestro código de prueba…

```
@pytest.mark.parametrize("a, b", [
    ("", 5),  # empty
    (87, "foo"),  # not a number
])
def test_with_letters_bad_value(a, b):
    with pytest.raises(ValueError):
        adder(a, b)
```

...y volvemos a ejecutar:

```
unittesting/v04 $ fades -d pytest -x pytest -q test_adder.py
......FFFF                                                               [100%]
(...)
```

Ahora sí, todas fallas, esperables porque todavía no trabajamos el código de nuestra función. Es sencillo, sólo tenemos que convertir lo recibido a número:

```
1 def adder(n1, n2):
2     return int(n1) + int(n2)
```

```
unittesting/v04 $ fades -d pytest -x pytest -q test_adder.py
..........                                                               [100%]
10 passed in 0.01s
```

Todo terminado. Ponemos nuestra función en producción, la incorporamos al sistema, distribuimos el código de alguna manera, o lo que sea que fuese necesario para que la gente use nuestro código. Y tiempo después sucede lo esperado: alguien encuentra un bug. Recibimos un reporte que indica que si ejecutan la función con `"2.1"` y `"3"`, el resultado no es el que corresponde, (`5.1`), sino `5`.

Analizando nuestra función encontramos que el problema es que estamos convirtiendo las cadenas a números enteros. Antes de intentar cualquier corrección, el primer paso es reproducir el bug con un caso de prueba (en realidad agregamos un caso a la parametrización de `test_with_letters_ok`):

```
19 @pytest.mark.parametrize("a, b, c", [
20     ("3", 5, 8),
21     (87, "-87", 0),
22     ("2.1", 3, 5.1),
23 ])
24 def test_with_letters_ok(a, b, c):
25     result = adder(a, b)
26     assert result == c
```

Como sólo modificamos una de las funciones, ejecutemos esas pruebas nada más, usando el parámetro `-k` de `pytest`:

```
unittesting/v05 $ fades -d pytest -x pytest -q test_adder.py -k test_with_letters_ok
..F                                                                      [100%]
(...)
FAILED test_adder.py::test_with_letters_ok[2.1-3-5.1] - ValueError: invalid literal for int() with base
↪  10: '2.1'
1 failed, 2 passed, 8 deselected in 0.01s
```

Vemos como `pytest` ejecutó tres pruebas (porque esa función es parametrizada), y nos termina indicando que falló un caso, pasaron exitosamente dos, y ocho más fueron deseleccionados (son parte del conjunto de pruebas pero no fueron incluidos en el filtro que especificamos).

Es hora de mejorar el código para soportar las nuevas cadenas:

```
def adder(n1, n2):
    return float(n1) + float(n2)
```

Y de paso agregamos varios casos de prueba más para puntos flotantes. Acá tenemos que tener la preocupación particular de los puntos flotantes binarios donde no es tan sencillo comparar por igualdad (como vimos en **??**). Para ello aprovechamos una funcionalidad de `pytest` para comparar si dos números son aproximadamente iguales...

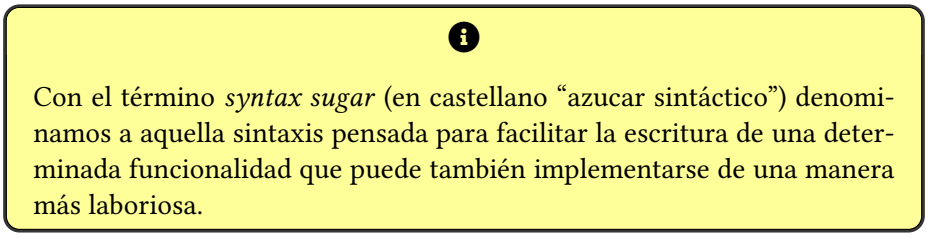
```
@pytest.mark.parametrize("a, b, c", [
    (0, 1.0, 1.0),
    (-5, 3.5, -1.5),
    (2.3, -0.1, 2.2),
])
def test_floating_point(a, b, c):
    result = adder(a, b)
    assert result == pytest.approx(c)
```

```
unittesting/v04 $ fades -d pytest -x pytest -q test_adder.py
..............                                                [100%]
14 passed in 0.02s
```

Y de esta manera terminamos los cambios en esta iteración, fuente del bug reportado. Seguramente nuestro código recibirá cambios en el futuro, productos de correción de errores o agregado de nuevas características. Los casos de prueba seguirán creciendo y cubriendo toda la funcionalidad que tenemos, de manera de poder asegurarnos que un cambio en el futuro no rompe alguna condición del pasado (de la cual es probable que nos hayamos olvidado), logrando de esta manera que nuestro sistema pueda evolucionar a través del tiempo con la calidad deseada.

1.5. Decoradores

Los decoradores son una *syntax sugar* de Python para reemplazar la definición de una función por otra generada dinámicamente.



Con el término *syntax sugar* (en castellano "azucar sintáctico") denominamos a aquella sintaxis pensada para facilitar la escritura de una determinada funcionalidad que puede también implementarse de una manera más laboriosa.

Antes de entrar en la funcionalidad propiamente dicha, presentaremos un ejemplo de cómo podría ser útil esa funcionalidad, para poder entenderla mejor.

Supongamos que tenemos un sistema con varias funciones de todo estilo y queremos imprimir por pantalla los parámetros que recibe cada función antes de ejecutarse y su resultado luego de correrse. Escribamos una función sencilla pero pensemos para el ejemplo que tendríamos muchas funciones con diversas complejidades.

CELL 01

```
def adder(n1, n2):
    return n1 + n2

res = adder(5, 6)
```

Por supuesto, podríamos modificar cada una de nuestras funciones y poner *prints* dentro de cada una, algo como lo siguiente:

CELL 02

```
def adder(n1, n2):
    print(f"Entrada de 'adder': {n1} {n2}")
    result = n1 + n2
    print(f"Resultado: {result}")
    return result

res = adder(5, 6)  # ejemplo de código que no queremos tocar
```

```
Entrada de 'adder': 5 6
Resultado: 11
```

Pero en realidad no es la mejor manera: no sólo sería mucho trabajo al tener que modificar todas las funciones una por una, sino que también tendríamos bastante código duplicado en todas las funciones. Por otro lado, esta solución laboriosa tiene la ventaja que no tenemos que modificar todo el código que usa nuestras funciones; en otras palabras, no queremos modificar la última línea de nuestro ejemplo `res = adder(5, 6)`.

Podemos realizar la tarea automáticamente con otra función.

Esta nueva función "impresora" realizará los prints necesarios: el de entrada antes de llamar a la función, y el de salida con el resultado de la misma. Para no tener que modificar todo el código que usa nuestra función original, lo que hacemos es reemplazar esa función original por otra que devuelve la función impresora.

CELL 03

```
def impresora(func_original):

    def nueva_func(*args, **kwargs):
        print(f"Entrada de '{func_original.__name__}': {args} {kwargs}")
        result = func_original(*args, **kwargs)
        print(f"Resultado: {result}")
        return result

    return nueva_func


def adder(n1, n2):  # función original, no la modificamos
    return n1 + n2

adder = impresora(adder)  # reemplazamos la función original

res = adder(5, 6)  # evitamos tocar el código que usa la función
```

```
Entrada de 'adder': (5, 6) {}
Resultado: 11
```

Antes de entrar en el detalle de la función "impresora" prestemos atención en cómo la usamos: la ejecutamos pasándole la función `adder` que acabamos de definir, y vinculamos su resultado con el mismo nombre que esa función. Es clave entender que en todo el resto del código (la última linea en nuestro ejemplo) cada vez que se use la función `adder` no se estará corriendo la función definida originalmente sino una función creada y devuelta por `impresora`.

La función `impresora` definida arriba recibirá entonces la función original y creará una nueva función (que es la que devuelve). Esta nueva función...

- recibirá cualquier combinación de parámetros (a través de `*args` y `**kwargs`, como explicamos en ??)
- mostrará por pantalla el nombre de la función y los parámetros recibidos
- ejecutará la función original, pasando los parámetros recibidos y guardando su resultado
- mostrará por pantalla ese resultado
- devolverá finalmente ese mismo resultado

En otras palabras, la nueva función definida por `impresora` (que es la que reemplaza a `adder`) tendrá el mismo comportamiento que esa función original, con el agregado de imprimir los parámetros recibidos antes de la ejecución y el resultado después.

Dejamos como tarea para les lectores el adaptar la función `impresora` para soportar el caso de que la función original genere una excepción, ante lo cual debería imprimir por pantalla la excepción ocurrida y luego dejarla continuar.

Destaquemos que la función nueva es del todo genérica: usará la función original y hará de pasamanos de los parámetros recibidos y el resultado final. De esta manera la función `impresora` se podrá usar para reemplazar cualquier función con cualquier cantidad de parámetros. Por ejemplo:

CELL 04

```
def square_root(num):
    return num ** 0.5

square_root = impresora(square_root)

res = square_root(25)
```

```
Entrada de 'square_root': (25,) {}
Resultado: 5.0
```

El azucar sintáctico que nos agrega Python sobre esta funcionalidad es la capacidad de redefinir dinámicamente la función definida en el código de forma simple. Entonces en vez de reemplazar cada función con la llamada manual a `impresora`, se hace todo automáticamente decorando la función definida:

CELL 05

```
@impresora
def adder(n1, n2):
    res = n1 + n2
    return res


@impresora
def square_root(num):
    return num ** 0.5


print("Arrancamos")
res1 = adder(5, 6)
res2 = square_root(25)
print(f"Los resultados son {res1} y {res2}")
```

```
Arrancamos
Entrada de 'adder': (5, 6) {}
Resultado: 11
Entrada de 'square_root': (25,) {}
Resultado: 5.0
Los resultados son 11 y 5.0
```

Notemos que al usar la sintaxis del `@` lo único que tenemos que especificar es la función decoradora. Podemos extender esta sintaxis para parametrizar nuestro decorador. Para mejorar nuestro ejemplo, podríamos indicarle un prefijo a la función impresora. Entonces, en vez de decorar con `@impresora`, decoraríamos por ejemplo con `@impresora("=")`. Lo que sucede allí es que en lugar de especificar el `@` directamente con la función decoradora, ejecutamos una función que tiene que devolver la función decoradora.

CELL 06

```
def impresora(prefijo):

    def decorador(func_original):

        def nueva_func(*args, **kwargs):
            print(f"{prefijo * 10} Entrada de 'func_original.__name__': {args} {kwargs}")
            res = func_original(*args, **kwargs)
            print(f"{prefijo * 10} Resultado: {res}")
            return res

        return nueva_func

    return decorador
```

¡Vemos que ahora tenemos tres funciones anidadas! La primera es la que usamos con el `@`, que devuelve el decorador en sí. Luego la función decoradora adentro define otra función como veníamos entendiendo antes, no hay sorpresas en ese punto. El único cambio ahí dentro es que los prints ahora usan el prefijo que recibió `impresora`.

CELL 07

```
@impresora("=")
def adder(n1, n2):  # función original, no la modificamos
    res = n1 + n2
    return res


@impresora("-")
def square_root(num):
    return num ** 0.5


print("Arrancamos")
res1 = adder(5, 6)
res2 = square_root(25)
print(f"Los resultados son {res1} y {res2}")
```

```
Arrancamos
========== Entrada de 'func_original.__name__': (5, 6) {}
========== Resultado: 11
---------- Entrada de 'func_original.__name__': (25,) {}
---------- Resultado: 5.0
Los resultados son 11 y 5.0
```

Vemos finalmente que podemos modificar el comportamiento de nuestro decorador.

Parte II
Herramientas fundamentales

Los capítulos siguientes desarrollan un conjunto de conocimientos sobre herramientas fundamentales que serán aprovechados en la Parte III al abordar temas de aplicaciones específicas.

Parte III
Temas específicos

En esta Parte desarrollaremos capítulos que abordan temas específicos de aplicación de herramientas de Python. Cada capítulo está autocontenido, de forma que el lector o lectora puede acceder directamente al tema de interés sin realizar un recorrido secuencial de los capítulos, aunque serán utilizados conceptos y herramientas tratados en la Parte II.

Parte IV
Apéndices

A | Zen de Python

Incluimos aquí las frases traducidas correspondientes al Zen de Python [**zen-de-python**].

- Bello es mejor que feo.
- Explícito es mejor que implícito.
- Simple es mejor que complejo.
- Complejo es mejor que complicado.
- Plano es mejor que anidado.
- Espaciado es mejor que denso.
- La legibilidad es importante.
- Los casos especiales no son lo suficientemente especiales como para romper las reglas.
- Sin embargo la practicidad le gana a la pureza.
- Los errores nunca deberían pasar silenciosamente.
- A menos que se silencien explícitamente.
- Frente a la ambigüedad, evitar la tentación de adivinar.
- Debería haber una, y preferiblemente solo una, manera obvia de hacerlo.
- A pesar de que esa manera no sea obvia a menos que seas Holandés.
- Ahora es mejor que nunca.
- A pesar de que nunca es muchas veces mejor que *ahora* mismo.
- Si la implementación es difícil de explicar, es una mala idea.
- Si la implementación es fácil de explicar, puede que sea una buena idea.
- Los espacios de nombres son una gran idea, ¡tengamos más de esos!