# 11 Probar tu código



Cuando escribes una función o una clase, también puedes escribir pruebas para ese código. Las pruebas demuestran que tu código funciona como se supone que debe hacerlo en respuesta a todos los tipos de entrada que está diseñado para recibir. Cuando escribes pruebas, puedes estar seguro de que tu código funcionará correctamente cuando más gente empiece a utilizar tus programas. También podrás probar el código nuevo a medida que lo añadas, para asegurarte de que los cambios no rompen el comportamiento existente del programa. Todo programador comete errores, por lo que debe probar su código a menudo, para detectar los problemas antes de que los usuarios se encuentren con ellos.

En este capítulo, aprenderás a probar tu código utilizando pytest. La biblioteca pytest es una colección de herramientas que te ayudarán a escribir tus primeras pruebas de forma rápida y sencilla, a la vez que apoyan tus pruebas a medida que crecen en complejidad junto con tus proyectos. Python no incluye pytest por defecto, así que aprenderás a instalar bibliotecas externas. Saber instalar bibliotecas externas pondrá a tu disposición una gran variedad de código bien diseñado. Estas bibliotecas ampliarán enormemente el tipo de proyectos en los que puedes trabajar.

Aprenderás a construir una serie de pruebas y a comprobar que cada conjunto de entradas da como resultado la salida que deseas. Verás qué aspecto tiene una prueba superada y qué aspecto tiene una prueba fallida, y aprenderás cómo una prueba fallida puede ayudarte a mejorar tu código. Aprenderás a probar funciones y clases, y empezarás a comprender cuántas pruebas debes escribir para un proyecto.

## Instalar pytest con pip

Aunque Python incluye mucha funcionalidad en la biblioteca estándar, los desarrolladores de Python también dependen en gran medida de paquetes de terceros. Un *third-party package* es una biblioteca que se desarrolla fuera del núcleo del lenguaje Python. Algunas bibliotecas populares de terceros acaban siendo adoptadas por la biblioteca estándar, y terminan incluyéndose en la mayoría de las instalaciones de Python a partir de ese momento. Esto ocurre sobre todo con las bibliotecas que probablemente no cambien mucho una vez resueltos sus errores iniciales. Este tipo de bibliotecas pueden evolucionar al mismo ritmo que el lenguaje en general.

Muchos paquetes, sin embargo, se mantienen fuera de la biblioteca estándar para que puedan desarrollarse a un ritmo independiente del propio lenguaje. Estos paquetes tienden a actualizarse con más frecuencia de lo que lo harían si estuvieran ligados al calendario de desarrollo de Python. Este es el caso de pytest y de la mayoría de las bibliotecas que utilizaremos en la segunda mitad de este libro. No debes confiar ciegamente en todos los paquetes de terceros, pero tampoco debes desanimarte por el hecho de que muchas funcionalidades importantes se implementan a través de dichos paquetes.

### Actualizar pip

Python incluye una herramienta llamada pip que se utiliza para instalar paquetes de terceros. Como pip ayuda a instalar paquetes de recursos externos, se actualiza a menudo para solucionar posibles problemas de seguridad. Así que empezaremos por actualizar pip.

Abre una nueva ventana de terminal y ejecuta el siguiente comando:

$ python -m pip install --upgrade pip  
❶ Requirement already satisfied: pip in /.../python3.11/site-packages (22.0.4)  
--snip--  
❷ Successfully installed pip-22.1.2

La primera parte de este comando, python -m pip, indica a Python que ejecute el módulo pip. La segunda parte, install --upgrade, indica a pip que actualice un paquete ya instalado. La última parte, pip, especifica qué paquete de terceros debe actualizarse. La salida muestra que mi versión actual de pip, la 22.0.4 ❶, fue sustituida por la última versión en el momento de escribir esto, la 22.1.2 ❷.

Puedes utilizar este comando para actualizar cualquier paquete de terceros instalado en tu sistema:

$ python -m pip install --upgrade package\_name

## Nota

Si utilizas Linux, es posible que pip no esté incluido en tu instalación de Python. Si obtienes un error al intentar actualizar pip, consulta las instrucciones del Apéndice A.

### Instalar pytest

Ahora que pip está actualizado, podemos instalar pytest:

$ python -m pip install --user pytest  
Collecting pytest  
 --snip--  
Successfully installed attrs-21.4.0 iniconfig-1.1.1 ...pytest-7.x.x

Seguimos utilizando el comando principal pip install, sin la bandera --upgrade esta vez. En su lugar, utilizaremos la bandera --user, que indica a Python que instale este paquete sólo para el usuario actual. La salida muestra que se ha instalado correctamente la última versión de pytest, junto con otros paquetes de los que depende pytest.

Puedes utilizar este comando para instalar muchos paquetes de terceros:

$ python -m pip install --user package\_name

## Nota

Si tienes alguna dificultad para ejecutar este comando, intenta ejecutar el mismo comando sin la bandera --user.

## Probar una función

Para aprender a probar, necesitamos código que probar. He aquí una función sencilla que recibe un nombre y un apellido y devuelve un nombre completo con un formato claro:

**name\_function.py**

def get\_formatted\_name(first, last):  
 """Generate a neatly formatted full name."""  
 full\_name = f"{first} {last}"  
 return full\_name.title()

La función get\_formatted\_name() combina el nombre y los apellidos con un espacio intermedio para completar un nombre completo, y luego escribe en mayúsculas y devuelve el nombre completo. Para comprobar que get\_formatted\_name() funciona, vamos a crear un programa que utilice esta función. El programa *names.py* permite a los usuarios introducir un nombre y un apellido, y ver un nombre completo formateado ordenadamente:

**names.py**

from name\_function import get\_formatted\_name  
  
print("Enter 'q' at any time to quit.")  
while True:  
 first = input("\nPlease give me a first name: ")  
 if first == 'q':  
 break  
 last = input("Please give me a last name: ")  
 if last == 'q':  
 break  
  
 formatted\_name = get\_formatted\_name(first, last)  
 print(f"\tNeatly formatted name: {formatted\_name}.")

Este programa importa get\_formatted\_name() de *name\_function.py*. El usuario puede introducir una serie de nombres y apellidos y ver los nombres completos formateados que se generan:

Enter 'q' at any time to quit.  
  
Please give me a first name: janis  
Please give me a last name: joplin  
 Neatly formatted name: Janis Joplin.  
  
Please give me a first name: bob  
Please give me a last name: dylan  
 Neatly formatted name: Bob Dylan.  
  
Please give me a first name: q

Podemos ver que los nombres generados aquí son correctos. Pero digamos que queremos modificar get\_formatted\_name() para que también pueda manejar segundos nombres. Al hacerlo, queremos asegurarnos de que no rompemos la forma en que la función gestiona los nombres que sólo tienen nombre y apellidos. Podríamos probar nuestro código ejecutando *names.py* e introduciendo un nombre como Janis Joplin cada vez que modificamos get\_formatted\_name(), pero eso se volvería tedioso. Afortunadamente, pytest proporciona una forma eficaz de automatizar la comprobación de la salida de una función. Si automatizamos las pruebas de get\_formatted\_name(), siempre podremos estar seguros de que la función funcionará cuando se le den los tipos de nombres para los que hemos escrito pruebas.

### Pruebas unitarias y casos de prueba

Existe una gran variedad de enfoques para probar el software. Uno de los tipos de prueba más sencillos es la prueba unitaria. Una *unit test* verifica que un aspecto concreto del comportamiento de una función es correcto. Un *test case* es una colección de pruebas unitarias que, en conjunto, demuestran que una función se comporta como se supone que debe hacerlo, dentro de toda la gama de situaciones que esperas que maneje.

Un buen caso de prueba tiene en cuenta todos los tipos posibles de entrada que podría recibir una función e incluye pruebas para representar cada una de estas situaciones. Un caso de prueba con *full coverage* incluye una gama completa de pruebas unitarias que cubren todas las formas posibles de utilizar una función. Conseguir una cobertura completa en un proyecto grande puede ser desalentador. A menudo es suficiente con escribir pruebas para los comportamientos críticos de tu código y luego aspirar a una cobertura total sólo si el proyecto empieza a tener un uso generalizado.

### Una prueba que pasa

Con pytest, escribir tu primera prueba unitaria es bastante sencillo. Escribiremos una única función de prueba. La función de prueba llamará a la función que estamos probando y haremos una afirmación sobre el valor devuelto. Si nuestra afirmación es correcta, la prueba pasará; si la afirmación es incorrecta, la prueba fallará.

Aquí tienes la primera prueba de la función get\_formatted\_name():

**test\_name\_function.py**

from name\_function import get\_formatted\_name  
  
❶ def test\_first\_last\_name():  
 """Do names like 'Janis Joplin' work?"""  
❷ formatted\_name = get\_formatted\_name('janis', 'joplin')  
❸ assert formatted\_name == 'Janis Joplin'

Antes de ejecutar la prueba, echemos un vistazo más de cerca a esta función. El nombre del archivo de prueba es importante; debe empezar por *test\_*. Cuando le pidamos a pytest que ejecute las pruebas que hemos escrito, buscará cualquier archivo que empiece por *test\_*, y ejecutará todas las pruebas que encuentre en ese archivo.

En el archivo de prueba, primero importamos la función que queremos probar: get\_formatted\_name(). Después definimos una función de prueba: en este caso, test\_first\_last\_name() ❶. Se trata de un nombre de función más largo que el que hemos estado utilizando, por una buena razón. En primer lugar, las funciones de prueba deben empezar por la palabra *test*, seguida de un guión bajo. Cualquier función que empiece por test\_ será *discovered* por pytest, y se ejecutará como parte del proceso de prueba.

Además, los nombres de las pruebas deben ser más largos y descriptivos que el nombre típico de una función. Nunca llamarás a la función tú mismo; pytest encontrará la función y la ejecutará por ti. Los nombres de las funciones de prueba deben ser lo suficientemente largos como para que, si ves el nombre de la función en un informe de prueba, tengas una buena idea de qué comportamiento se estaba probando.

A continuación, llamamos a la función que estamos probando ❷. Aquí llamamos a get\_formatted\_name() con los argumentos 'janis' y 'joplin', igual que utilizamos cuando ejecutamos *names.py*. Asignamos el valor de retorno de esta función a formatted\_name.

Por último, hacemos una afirmación ❸. Una *assertion* es una afirmación sobre una condición. Aquí afirmamos que el valor de formatted\_name debe ser 'Janis Joplin'.

### Ejecutar una prueba

Si ejecutas directamente el archivo *test\_name\_function.py*, no obtendrás ninguna salida porque nunca hemos llamado a la función de prueba. En su lugar, haremos que pytest ejecute el archivo de prueba por nosotros.

Para ello, abre una ventana de terminal y navega hasta la carpeta que contiene el archivo de prueba. Si utilizas VS Code, puedes abrir la carpeta que contiene el archivo de prueba y utilizar el terminal que está incrustado en la ventana del editor. En la ventana del terminal, introduce el comando pytest. Esto es lo que deberías ver:

$ pytest  
========================= test session starts =========================  
❶ platform darwin -- Python 3.x.x, pytest-7.x.x, pluggy-1.x.x  
❷ rootdir: /.../python\_work/chapter\_11  
❸ collected 1 item  
  
❹ test\_name\_function.py . [100%]  
========================== 1 passed in 0.00s ==========================

Intentemos dar sentido a esta salida. En primer lugar, vemos algo de información sobre el sistema en el que se está ejecutando la prueba ❶. Lo estoy probando en un sistema macOS, por lo que es posible que veas algunos resultados diferentes. Lo más importante es que podemos ver qué versiones de Python, pytest y otros paquetes se están utilizando para ejecutar la prueba.

A continuación, vemos el directorio desde el que se está ejecutando la prueba ❷: en mi caso, *python\_work/chapter\_11*. Podemos ver que pytest ha encontrado una prueba para ejecutar ❸, y podemos ver el archivo de prueba que se está ejecutando ❹. El único punto tras el nombre del archivo nos indica que se ha superado una única prueba, y el 100% deja claro que se han ejecutado todas las pruebas. Un proyecto grande puede tener cientos o miles de pruebas, y los puntos y el indicador de porcentaje completado pueden ser útiles para controlar el progreso general de la ejecución de las pruebas.

La última línea nos indica que se ha superado una prueba y que se ha tardado menos de 0,01 segundos en ejecutarla.

Esta salida indica que la función get\_formatted\_name() siempre funcionará para nombres que tengan nombre y apellidos, a menos que modifiquemos la función. Cuando modifiquemos get\_formatted\_name(), podremos volver a ejecutar esta prueba. Si la prueba se supera, sabremos que la función seguirá funcionando para nombres como Janis Joplin.

## Nota

Si no estás seguro de cómo navegar hasta la ubicación correcta en el terminal, consulta "Ejecutar programas Python desde un terminal" en la página 11. Además, si aparece un mensaje que indica que no se ha encontrado el comando pytest, utiliza en su lugar el comando python -m pytest.

### Una prueba fallida

¿Qué aspecto tiene una prueba que falla? Modifiquemos get\_formatted\_name() para que pueda manejar segundos nombres, pero hagámoslo de forma que rompa la función para nombres con sólo nombre y apellidos, como Janis Joplin.

Aquí tienes una nueva versión de get\_formatted\_name() que requiere un argumento de segundo nombre:

**name\_function.py**

def get\_formatted\_name(first, middle, last):  
 """Generate a neatly formatted full name."""  
 full\_name = f"{first} {middle} {last}"  
 return full\_name.title()

Esta versión debería funcionar para las personas con segundo nombre, pero cuando la probamos, vemos que hemos roto la función para las personas que sólo tienen nombre y apellidos.

Esta vez, al ejecutar pytest obtenemos el siguiente resultado:

$ pytest  
========================= test session starts =========================  
--snip--  
❶ test\_name\_function.py F [100%]  
❷ ============================== FAILURES ===============================  
❸ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ test\_first\_last\_name \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
 def test\_first\_last\_name():  
 """Do names like 'Janis Joplin' work?"""  
❹ > formatted\_name = get\_formatted\_name('janis', 'joplin')  
❺ E TypeError: get\_formatted\_name() missing 1 required positional  
 argument: 'last'  
  
test\_name\_function.py:5: TypeError  
======================= short test summary info =======================  
FAILED test\_name\_function.py::test\_first\_last\_name - TypeError:  
 get\_formatted\_name() missing 1 required positional argument: 'last'  
========================== 1 failed in 0.04s ==========================

Hay mucha información aquí porque hay muchas cosas que puedes necesitar saber cuando falla una prueba. El primer elemento destacable de la salida es un único F ❶, que nos indica que ha fallado una prueba. A continuación vemos una sección que se centra en FAILURES ❷, porque las pruebas fallidas suelen ser lo más importante en lo que hay que centrarse en una ejecución de prueba. A continuación, vemos que test\_first\_last\_name() fue la función de prueba que falló ❸. Un corchete angular ❹ indica la línea de código que provocó el fallo de la prueba. El E de la línea siguiente ❺ muestra el error real que causó el fallo: un TypeError debido a la falta de un argumento posicional requerido, last. La información más importante se repite en un resumen más breve al final, de modo que cuando ejecutes muchas pruebas, puedas hacerte una idea rápida de qué pruebas fallaron y por qué.

### Responder a una prueba fallida

¿Qué haces cuando falla una prueba? Suponiendo que estés comprobando las condiciones correctas, una prueba superada significa que la función se comporta correctamente y una prueba fallida significa que hay un error en el nuevo código que has escrito. Así que cuando falle una prueba, no cambies la prueba. Si lo haces, puede que tus pruebas pasen, pero cualquier código que llame a tu función como lo hace la prueba dejará de funcionar de repente. En lugar de eso, arregla el código que está haciendo que falle la prueba. Examina los cambios que acabas de hacer en la función, y averigua cómo esos cambios rompieron el comportamiento deseado.

En este caso, get\_formatted\_name() solía requerir sólo dos parámetros: un nombre y un apellido. Ahora requiere un nombre, un segundo nombre y un apellido. La adición de ese parámetro obligatorio del segundo nombre rompió el comportamiento original de get\_formatted\_name(). La mejor opción en este caso es hacer que el segundo nombre sea opcional. Una vez que lo hagamos, nuestra prueba para nombres como Janis Joplin debería volver a pasar, y también deberíamos poder aceptar segundos nombres. Modifiquemos get\_formatted\_name() para que los segundos nombres sean opcionales y volvamos a ejecutar el caso de prueba. Si se supera, pasaremos a asegurarnos de que la función gestiona correctamente los segundos nombres.

Para que los segundos nombres sean opcionales, movemos el parámetro middle al final de la lista de parámetros en la definición de la función y le damos un valor por defecto vacío. También añadimos una prueba if que construye el nombre completo correctamente, dependiendo de si se proporciona un segundo nombre:

**name\_function.py**

def get\_formatted\_name(first, last, middle=''):  
 """Generate a neatly formatted full name."""  
 if middle:  
 full\_name = f"{first} {middle} {last}"  
 else:  
 full\_name = f"{first} {last}"  
 return full\_name.title()

En esta nueva versión de get\_formatted\_name(), el segundo nombre es opcional. Si se pasa un segundo nombre a la función, el nombre completo contendrá un nombre, un segundo nombre y un apellido. En caso contrario, el nombre completo constará sólo de nombre y apellidos. Ahora la función debería funcionar para ambos tipos de nombres. Para averiguar si la función sigue funcionando para nombres como Janis Joplin, vamos a ejecutar de nuevo la prueba:

$ pytest  
========================= test session starts =========================  
--snip--  
test\_name\_function.py . [100%]  
========================== 1 passed in 0.00s ==========================

Ahora la prueba pasa. Esto es ideal; significa que la función vuelve a funcionar para nombres como Janis Joplin, sin que tengamos que probar la función manualmente. Arreglar nuestra función fue más fácil porque la prueba fallida nos ayudó a identificar cómo el nuevo código rompía el comportamiento existente.

### Añadir nuevas pruebas

Ahora que sabemos que get\_formatted\_name() vuelve a funcionar con nombres sencillos, escribamos una segunda prueba para las personas que incluyen un segundo nombre. Para ello, añadimos otra función de prueba al archivo *test\_name\_function.py*:

**test\_name\_function.py**

from name\_function import get\_formatted\_name  
  
def test\_first\_last\_name():  
 --snip--  
  
def test\_first\_last\_middle\_name():  
 """Do names like 'Wolfgang Amadeus Mozart' work?"""  
❶ formatted\_name = get\_formatted\_name(  
 'wolfgang', 'mozart', 'amadeus')  
❷ assert formatted\_name == 'Wolfgang Amadeus Mozart'

Llamamos a esta nueva función test\_first\_last\_middle\_name(). El nombre de la función debe empezar por test\_ para que la función se ejecute automáticamente cuando ejecutemos pytest. Nombramos la función para que quede claro qué comportamiento de get\_formatted\_name() estamos probando. De este modo, si la prueba falla, sabremos inmediatamente qué tipo de nombres se ven afectados.

Para probar la función, llamamos a get\_formatted\_name() con un nombre, apellido y segundo nombre ❶, y luego hacemos una afirmación ❷ de que el nombre completo devuelto coincide con el nombre completo (nombre, segundo nombre y apellido) que esperamos. Cuando volvemos a ejecutar pytest, ambas pruebas se superan:

$ pytest  
========================= test session starts =========================  
--snip--  
collected 2 items  
  
❶ test\_name\_function.py .. [100%]  
========================== 2 passed in 0.01s ==========================

Los dos puntos ❶ indican que se han superado dos pruebas, lo que también queda claro en la última línea de salida. ¡Esto es estupendo! Ahora sabemos que la función sigue funcionando con nombres como Janis Joplin, y podemos estar seguros de que también funcionará con nombres como Wolfgang Amadeus Mozart.

## Pruébalo tú mismo

11-1. Ciudad, País: Escribe una función que acepte dos parámetros: el nombre de una ciudad y el nombre de un país. La función debe devolver una única cadena de la forma Ciudad, País, como Santiago, Chile. Guarda la función en un módulo llamado *city\_functions.py*, y guarda este archivo en una carpeta nueva para que pytest no intente ejecutar las pruebas que ya hemos escrito.

Crea un archivo llamado *test\_cities.py* que compruebe la función que acabas de escribir. Escribe una función llamada test\_city\_country() para verificar que al llamar a tu función con valores como 'santiago' y 'chile' se obtiene la cadena correcta. Ejecuta la prueba y asegúrate de que test\_city\_country() pasa.

11-2. Población: Modifica tu función para que requiera un tercer parámetro, population. Ahora debería devolver una única cadena de la forma Ciudad, País - población xxx, como Santiago, Chile – population 5000000. Ejecuta de nuevo la prueba y asegúrate de que test\_city\_country() falla esta vez.

Modifica la función para que el parámetro population sea opcional. Ejecuta la prueba, y asegúrate de que test\_city\_country() pasa de nuevo.

Escribe una segunda prueba llamada test\_city\_country\_population() que verifique que puedes llamar a tu función con los valores 'santiago', 'chile' y 'population=5000000'. Ejecuta las pruebas una vez más y asegúrate de que esta nueva prueba se supera.

## Probar una clase

En la primera parte de este capítulo, escribiste pruebas para una única función. Ahora escribirás pruebas para una clase. Utilizarás clases en muchos de tus programas, por lo que es útil poder demostrar que tus clases funcionan correctamente. Si tienes pruebas que pasan para una clase en la que estás trabajando, puedes estar seguro de que las mejoras que hagas en la clase no romperán accidentalmente su comportamiento actual.

### Variedad de afirmaciones

Hasta ahora sólo has visto un tipo de aserción: la afirmación de que una cadena tiene un valor determinado. Al escribir una prueba, puedes hacer cualquier afirmación que pueda expresarse como una declaración condicional. Si la condición es True como esperabas, se confirmará tu suposición sobre cómo se comporta esa parte de tu programa; puedes estar seguro de que no existen errores. Si la condición que supone que es True es en realidad False, la prueba fallará y sabrás que hay un problema que resolver. [La Tabla 11-1](#table11-1) muestra algunos de los tipos de aserciones más útiles que puedes incluir en tus pruebas iniciales.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | **Afirmación** | **Afirmación** | | --- | --- | | assert a == b | Afirma que dos valores son iguales. | | assert a != b | Afirma que dos valores no son iguales. | | assert a | Afirma que a es igual a True. | | assert not a | Afirma que a se evalúa como False. | | assert elemento in lista | Afirma que un elemento está en una lista. | | assert elemento not in lista | Afirma que un elemento no está en una lista. | |

Tabla 11-1: Afirmaciones de uso común en las pruebas

Estos son sólo algunos ejemplos; cualquier cosa que pueda expresarse como una declaración condicional puede incluirse en una prueba.

### Una clase para probar

Probar una clase es similar a probar una función, porque gran parte del trabajo consiste en probar el comportamiento de los métodos de la clase. Sin embargo, hay algunas diferencias, así que vamos a escribir una clase para probar. Considera una clase que ayude a administrar encuestas anónimas:

**survey.py**

class AnonymousSurvey:  
 """Collect anonymous answers to a survey question."""  
  
❶ def \_\_init\_\_(self, question):  
 """Store a question, and prepare to store responses."""  
 self.question = question  
 self.responses = []  
  
❷ def show\_question(self):  
 """Show the survey question."""  
 print(self.question)  
  
❸ def store\_response(self, new\_response):  
 """Store a single response to the survey."""  
 self.responses.append(new\_response)  
  
❹ def show\_results(self):  
 """Show all the responses that have been given."""  
 print("Survey results:")  
 for response in self.responses:  
 print(f"- {response}")

Esta clase comienza con una pregunta de encuesta que tú proporcionas ❶ e incluye una lista vacía para almacenar las respuestas. La clase tiene métodos para imprimir la pregunta de la encuesta ❷, añadir una nueva respuesta a la lista de respuestas ❸, e imprimir todas las respuestas almacenadas en la lista ❹. Para crear una instancia a partir de esta clase, todo lo que tienes que proporcionar es una pregunta. Una vez que tengas una instancia que represente una encuesta concreta, muestra la pregunta de la encuesta con show\_question(), almacena una respuesta utilizando store\_response(), y muestra los resultados con show\_results().

Para demostrar que la clase AnonymousSurvey funciona, vamos a escribir un programa que utilice la clase:

**language\_survey.py**

from survey import AnonymousSurvey  
  
# Define a question, and make a survey.  
question = "What language did you first learn to speak?"  
language\_survey = AnonymousSurvey(question)  
  
# Show the question, and store responses to the question.  
language\_survey.show\_question()  
print("Enter 'q' at any time to quit.\n")  
while True:  
 response = input("Language: ")  
 if response == 'q':  
 break  
 language\_survey.store\_response(response)  
  
# Show the survey results.  
print("\nThank you to everyone who participated in the survey!")  
language\_survey.show\_results()

Este programa define una pregunta ("What language did you first learn to speak?") y crea un objeto AnonymousSurvey con esa pregunta. El programa llama a show\_question() para mostrar la pregunta y luego pide respuestas. Cada respuesta se almacena a medida que se recibe. Cuando se han introducido todas las respuestas (el usuario introduce q para salir), show\_results() imprime los resultados de la encuesta:

What language did you first learn to speak?  
Enter 'q' at any time to quit.  
  
Language: English  
Language: Spanish  
Language: English  
Language: Mandarin  
Language: q  
  
Thank you to everyone who participated in the survey!  
Survey results:  
- English  
- Spanish  
- English  
- Mandarin

Esta clase funciona para una simple encuesta anónima, pero supongamos que queremos mejorar AnonymousSurvey y el módulo en el que se encuentra, survey. Podríamos permitir que cada usuario introdujera más de una respuesta, podríamos escribir un método para listar sólo las respuestas únicas e informar de cuántas veces se ha dado cada respuesta, o incluso podríamos escribir otra clase para gestionar las encuestas no anónimas.

Implementar tales cambios podría afectar al comportamiento actual de la clase AnonymousSurvey. Por ejemplo, es posible que, al intentar permitir que cada usuario introduzca varias respuestas, cambiemos accidentalmente el modo en que se gestionan las respuestas únicas. Para asegurarnos de que no rompemos el comportamiento existente mientras desarrollamos este módulo, podemos escribir pruebas para la clase.

### Prueba de la clase EncuestaAnónima

Vamos a escribir una prueba que verifique un aspecto del comportamiento de AnonymousSurvey. Escribiremos una prueba para verificar que una única respuesta a la pregunta de la encuesta se almacena correctamente:

**test\_survey.py**

from survey import AnonymousSurvey  
  
❶ def test\_store\_single\_response():  
 """Test that a single response is stored properly."""  
 question = "What language did you first learn to speak?"  
❷ language\_survey = AnonymousSurvey(question)  
 language\_survey.store\_response('English')  
❸ assert 'English' in language\_survey.responses

Empezaremos importando la clase que queremos probar, AnonymousSurvey. La primera función de prueba verificará que cuando almacenemos una respuesta a la pregunta de la encuesta, la respuesta acabará en la lista de respuestas de la encuesta. Un buen nombre descriptivo para esta función es test\_store\_single\_response() ❶. Si esta prueba falla, sabremos por el nombre de la función en el resumen de la prueba que hubo un problema al almacenar una única respuesta a la encuesta.

Para probar el comportamiento de una clase, necesitamos crear una instancia de la clase. Creamos una instancia llamada language\_survey ❷ con la pregunta "What language did you first learn to speak?" Almacenamos una única respuesta, English, utilizando el método store\_response(). A continuación, verificamos que la respuesta se ha almacenado correctamente afirmando que English está en la lista language\_survey.responses ❸.

Por defecto, la ejecución del comando pytest sin argumentos ejecutará todas las pruebas que pytest descubra en el directorio actual. Para centrarte en las pruebas de un archivo, pasa el nombre del archivo de pruebas que quieras ejecutar. Aquí ejecutaremos sólo la prueba que escribimos para AnonymousSurvey:

$ pytest test\_survey.py  
========================= test session starts =========================  
--snip--  
test\_survey.py . [100%]  
========================== 1 passed in 0.01s ==========================

Es un buen comienzo, pero una encuesta sólo es útil si genera más de una respuesta. Verifiquemos que se pueden almacenar tres respuestas correctamente. Para ello, añadimos otro método a TestAnonymousSurvey:

from survey import AnonymousSurvey  
  
def test\_store\_single\_response():  
 --snip--  
  
def test\_store\_three\_responses():  
 """Test that three individual responses are stored properly."""  
 question = "What language did you first learn to speak?"  
 language\_survey = AnonymousSurvey(question)  
❶ responses = ['English', 'Spanish', 'Mandarin']  
 for response in responses:  
 language\_survey.store\_response(response)  
  
❷ for response in responses:  
 assert response in language\_survey.responses

Llamamos a la nueva función test\_store\_three\_responses(). Creamos un objeto encuesta igual que hicimos en test\_store\_single\_response(). Definimos una lista que contenga tres respuestas ❶ diferentes , y luego llamamos a store\_response() para cada una de estas respuestas. Una vez almacenadas las respuestas, escribimos otro bucle y afirmamos que cada respuesta está ahora en language\_survey.responses ❷.

Cuando volvemos a ejecutar el archivo de prueba, ambas pruebas (para una sola respuesta y para tres respuestas) pasan:

$ pytest test\_survey.py  
========================= test session starts =========================  
--snip--  
test\_survey.py .. [100%]  
========================== 2 passed in 0.01s ==========================

Esto funciona perfectamente. Sin embargo, estas pruebas son un poco repetitivas, así que utilizaremos otra función de pytest para hacerlas más eficientes.

### Utilizar dispositivos

En *test\_survey.py*, creamos una nueva instancia de AnonymousSurvey en cada función de prueba. Esto está bien en el breve ejemplo con el que estamos trabajando, pero en un proyecto del mundo real con decenas o cientos de pruebas, esto sería problemático.

En las pruebas, un *fixture* ayuda a configurar un entorno de pruebas. A menudo, esto significa crear un recurso que es utilizado por más de una prueba. Creamos un fixture en pytest escribiendo una función con el decorador @pytest.fixture. Un *decorator* es una directiva que se coloca justo antes de la definición de una función; Python aplica esta directiva a la función antes de que se ejecute, para alterar cómo se comporta el código de la función. No te preocupes si esto te parece complicado; puedes empezar a utilizar decoradores de paquetes de terceros antes de aprender a escribirlos tú mismo.

Utilicemos un fixture para crear una única instancia de encuesta que pueda utilizarse en ambas funciones de prueba en *test\_survey.py*:

import pytest  
from survey import AnonymousSurvey  
  
❶ @pytest.fixture  
❷ def language\_survey():  
 """A survey that will be available to all test functions."""  
 question = "What language did you first learn to speak?"  
 language\_survey = AnonymousSurvey(question)  
 return language\_survey  
  
❸ def test\_store\_single\_response(language\_survey):  
 """Test that a single response is stored properly."""  
❹ language\_survey.store\_response('English')  
 assert 'English' in language\_survey.responses  
  
❺ def test\_store\_three\_responses(language\_survey):  
 """Test that three individual responses are stored properly."""  
 responses = ['English', 'Spanish', 'Mandarin']  
 for response in responses:  
❻ language\_survey.store\_response(response)  
  
 for response in responses:  
 assert response in language\_survey.responses

Ahora necesitamos importar pytest, porque estamos utilizando un decorador que está definido en pytest. Aplicamos el decorador @pytest.fixture ❶ a la nueva función language\_survey() ❷. Esta función construye un objeto AnonymousSurvey y devuelve la nueva encuesta.

Observa que las definiciones de ambas funciones de prueba han cambiado ❸❺; cada función de prueba tiene ahora un parámetro llamado language\_survey. Cuando un parámetro de una función de prueba coincida con el nombre de una función con el decorador @pytest.fixture, el accesorio se ejecutará automáticamente y el valor devuelto se pasará a la función de prueba. En este ejemplo, la función language\_survey() suministra tanto a test\_store\_single\_response() como a test\_store\_three\_responses() una instancia de language\_survey.

No hay código nuevo en ninguna de las dos funciones de prueba, pero fíjate en que se han eliminado dos líneas de cada función ❹❻: la línea que definía una pregunta y la que creaba un objeto AnonymousSurvey.

Cuando volvemos a ejecutar el archivo de prueba, ambas pruebas siguen pasando. Estas pruebas serían especialmente útiles si intentáramos ampliar AnonymousSurvey para gestionar múltiples respuestas para cada persona. Después de modificar el código para que acepte múltiples respuestas, podrías ejecutar estas pruebas y asegurarte de que no has afectado a la capacidad de almacenar una sola respuesta o una serie de respuestas individuales.

Es casi seguro que la estructura anterior te parecerá complicada; contiene parte del código más abstracto que has visto hasta ahora. No es necesario que utilices fixtures de inmediato; es mejor escribir pruebas con mucho código repetitivo que no escribir ninguna prueba. Sólo tienes que saber que cuando hayas escrito suficientes pruebas como para que la repetición se interponga en tu camino, existe una forma bien establecida de tratar la repetición. Además, en ejemplos sencillos como éste, las fijaciones no hacen que el código sea más corto ni más fácil de seguir. Pero en proyectos con muchas pruebas, o en situaciones en las que se necesitan muchas líneas para construir un recurso que se utiliza en varias pruebas, las fijaciones pueden mejorar drásticamente tu código de prueba.

Cuando quieras escribir un fixture, escribe una función que genere el recurso que utilizan varias funciones de prueba. Añade el decorador @pytest.fixture  a la nueva función, y añade el nombre de esta función como parámetro para cada función de prueba que utilice este recurso. A partir de ese momento, tus pruebas serán más cortas y más fáciles de escribir y mantener.

## Pruébalo tú mismo

11-3. Empleado: Escribe una clase llamada Employee. El método \_\_init\_\_() debe tomar un nombre, un apellido y un salario anual, y almacenar cada uno de ellos como atributos. Escribe un método llamado give\_raise() que añada 5.000$ al salario anual por defecto, pero que también acepte una cantidad de aumento diferente.

Escribe un archivo de prueba para Employee con dos funciones de prueba, test\_give\_default\_raise() y test\_give\_custom\_raise(). Escribe tus pruebas una vez sin utilizar un fixture, y asegúrate de que ambas pasan. A continuación, escribe un fixture para no tener que crear una nueva instancia de empleado en cada función de prueba. Vuelve a ejecutar las pruebas y asegúrate de que ambas siguen siendo correctas.

## Resumen

En este capítulo has aprendido a escribir pruebas para funciones y clases utilizando las herramientas del módulo pytest. Has aprendido a escribir funciones de prueba que verifican comportamientos específicos que deben mostrar tus funciones y clases. Has visto cómo se pueden utilizar las fijaciones para crear eficientemente recursos que se pueden utilizar en múltiples funciones de prueba en un archivo de prueba.

Las pruebas son un tema importante al que muchos programadores noveles no están expuestos. No tienes que escribir pruebas para todos los proyectos sencillos que pruebes como nuevo programador. Pero en cuanto empieces a trabajar en proyectos que impliquen un esfuerzo de desarrollo importante, debes probar los comportamientos críticos de tus funciones y clases. Estarás más seguro de que el nuevo trabajo en tu proyecto no romperá las partes que funcionan, y esto te dará libertad para introducir mejoras en tu código. Si accidentalmente rompes una funcionalidad existente, lo sabrás enseguida, por lo que podrás solucionar el problema fácilmente. Responder a una prueba fallida que has ejecutado es mucho más fácil que responder a un informe de error de un usuario descontento.

Otros programadores respetarán más tus proyectos si incluyes algunas pruebas iniciales. Se sentirán más cómodos experimentando con tu código y estarán más dispuestos a trabajar contigo en los proyectos. Si quieres contribuir a un proyecto en el que están trabajando otros programadores, se esperará de ti que demuestres que tu código supera las pruebas existentes y, normalmente, se esperará de ti que escribas pruebas para cualquier comportamiento nuevo que introduzcas en el proyecto.

Juega con las pruebas para familiarizarte con el proceso de comprobación de tu código. Escribe pruebas para los comportamientos más críticos de tus funciones y clases, pero no pretendas una cobertura total en los primeros proyectos a menos que tengas una razón específica para hacerlo.