

# Introducción al Desarrollo de Software I - Lanzillotta

Capra - Chaves - Di Matteo - Sosa - Villegas - Palavecino

# Agenda

- Test Driven Development (TDD)
  - Instalación de un Framework de Testing
-

# Test Driven Development (TDD)

# ¿Qué es TDD?

- Proceso de desarrollo de software
- Convierte los Requisitos del Software en Casos de Prueba antes de que el software esté completamente desarrollado
- Las pruebas desarrolladas son ejecutadas repetitivamente
  - Es contrario a desarrollar primero el software y crear casos de prueba después

# Pasos a seguir para hacer TDD

1. Agregar una prueba
2. Correr todas las pruebas y ver que la nueva falla
3. Cambiar el código para que la prueba agregada efectivamente pase
4. Correr todas las pruebas y ver que todas se ejecutan correctamente
5. Refactorizar el código si es necesario

# Veamos un ejemplo: Un perro que ladra

## ❖ Caso de Uso:

- “Como perro quiero ladrar”

## ❖ Requerimiento:

- Todos los perros ingresados en el sistema deben poder ladrar



Vamos a embarrarnos un poco...

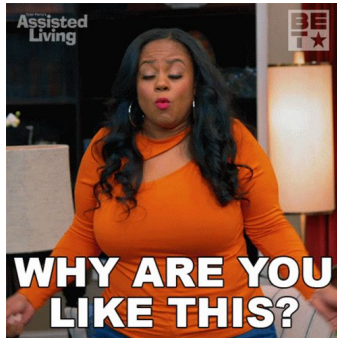
# 1) Agregar una prueba

```
class TestPerro(unittest.TestCase):  
    def test_perro_ladra(self):  
        perro = Perro()  
        self.assertEqual(perro.ladrar(), "Woof!")
```

¿así nomás?  
¡sí! ¡así nomás!

## 2) Correr todas las pruebas y ver que la nueva falla

¿Qué esperamos?  
¿Que todo se rompa!



```
def test_perro_ladra(self):  
    test_perro.TestPerro.test_perro_ladra failed with error: <class 'NameError'> name 'Perro' is not defined  
Traceback (most recent call last):  
  File "/Library/Developer/CommandLineTools/Library/Frameworks/Python3.framework/Versions/3.9/Lib/python3.9/unittest/case.py", line 59, in  
    yield  
  File "/Library/Developer/CommandLineTools/Library/Frameworks/Python3.framework/Versions/3.9/Lib/python3.9/unittest/case.py", line 593, in run  
    self._callTestMethod(testMethod)  
  File "/Library/Developer/CommandLineTools/Library/Frameworks/Python3.framework/Versions/3.9/Lib/python3.9/unittest/case.py", line 550, in  
    _callTestMethod  
    method()  
  File "/Users/cdimatteo/FIUBA/ids/test_perro.py", line 9, in test_perro_ladra  
    perro = Perro()  
NameError: name 'Perro' is not defined
```

test\_perro\_ladra (test\_perro.TestPerro) ... ERROR

ERROR: test\_perro\_ladra (test\_perro.TestPerro)

Traceback (most recent call last):  
 File "/Users/cdimatteo/FIUBA/ids/test\_perro.py", line 9, in test\_perro\_ladra  
 perro = Perro()  
NameError: name 'Perro' is not defined

Ran 1 test in 0.001s

FAILED (errors=1)  
Finished running tests!

veamos cuál es el paso que sigue



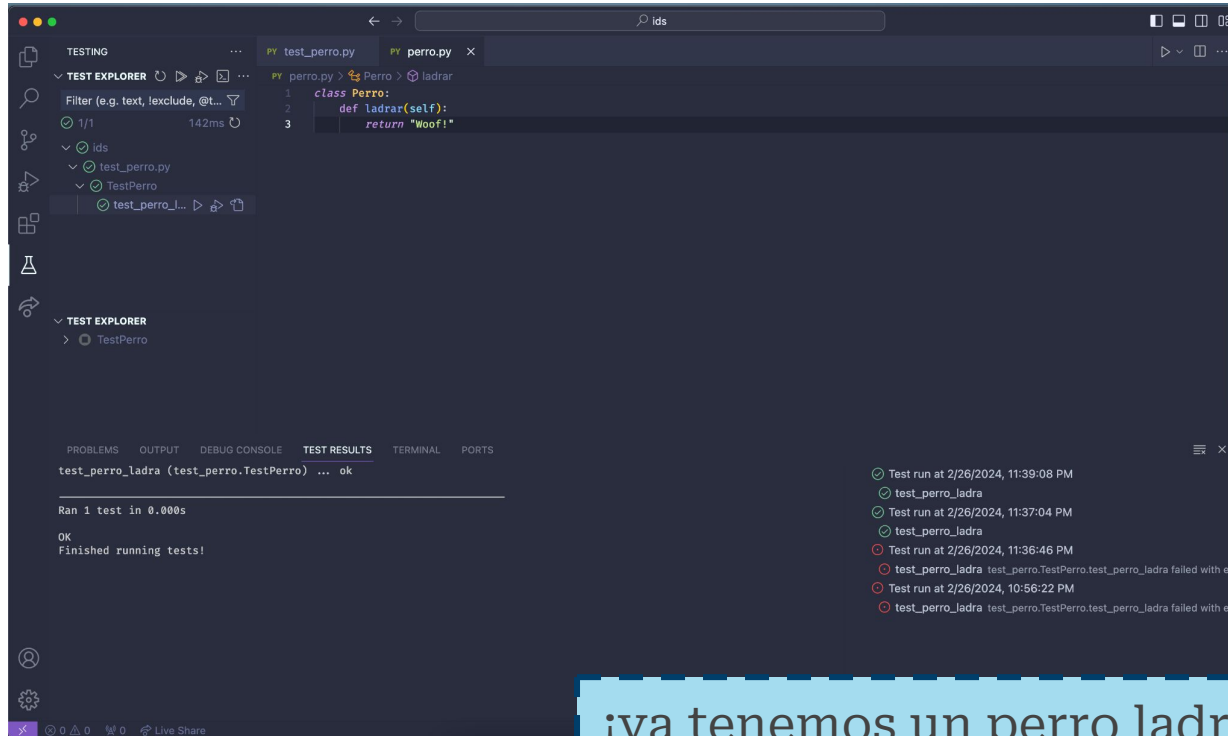


### 3) Cambiar el código para que la prueba agregada efectivamente pase

```
class Perro:  
    def ladrar(self):  
        return "Woof!"
```

¿y ahora?

## 4) Correr todas las pruebas y ver que todas se ejecutan correctamente



## 5) Refactorizar el código si es necesario

```
PY perro.py ×  
  
PY perro.py > Perro > ladrar  
1 class Perro:  
2     def ladrar(self):  
3         return "Woof!"
```

```
PY test_perro.py ×  
  
PY test_perro.py > ...  
1 import unittest  
2 from perro import Perro  
3  
4 class TestPerro(unittest.TestCase):  
5     def test_perro_ladra(self):  
6         perro = Perro()  
7         self.assertEqual(perro.ladrar(), "Woof!")  
8  
9 if __name__ == '__main__':  
10     unittest.main()  
11
```

En este caso, no tenemos código para refactorizar. Pero si lo hubiese, una vez corriendo las pruebas, es el momento para realizar la refactorización que sea necesaria.

# ¿Cómo se continúa el proceso?

A medida que el software crece en alcance (es decir, los requerimientos del usuario para con el mismo) deben agregarse como casos de prueba al flujo de desarrollo.

# Otros Casos de Uso para adicionar al ejemplo

## ❖ Casos de Uso:

1. “Como perro quiero pedir comida”
2. “Como perro quiero perseguir gatos”
3. “Como perro quiero correr”

## ❖ Requerimientos:

1. Todos los perros ingresados en el sistema deben poder pedir comida
2. Todos los perros ingresados en el sistema deben poder perseguir gatos
3. Todos los perros ingresados en el sistema deben poder correr

# 1) Agregamos las pruebas

```
class TestPerro(unittest.TestCase):  
    def test_perro_ladra(self):  
        perro = Perro()  
        self.assertEqual(perro.ladrar(), "Woof!")  
    def test_perro_pide_comida(self):  
        perro = Perro()  
        self.assertEqual(perro.pedir_comida(), "😭 🍖")  
    def test_perro_persigue_gatos(self):  
        perro = Perro()  
        self.assertEqual(perro.perseguir_gato(), "🐕 🌫️ 🐕")  
    def test_perro_corre(self):  
        perro = Perro()  
        self.assertEqual(perro.correr(), "🐕 🌫️")
```

## 2) Ejecutamos todas las pruebas

Como  
esperábamos,  
las nuevas  
rompen



### 3) Cambiamos el código

```
class Perro:
    def ladrar(self):
        return "Woof!"

    def pedir_comida(self):
        return "😭 🍖"

    def perseguir_gato(self):
        return "🐕💨 🐕"

    def correr(self):
        return "🐕💨"
```



## 4) Probamos nuevamente

```
✓ ✓ TestPerro
  ✓ test_perro_ladra
  ✓ test_perro_pide_comida
  ✓ test_perro_persigue_gatos
  ✓ test_perro_corre
```

y desarrollamos un perro  
que ladra, pide comida,  
persigue gatos y corre con  
TDD 😊

## 5) Refactorizamos código si es necesario

¡Pero no lo es! Así que dejamos al perro tranquilo.



# Camino al expertise

Ahora que ya entendimos el patrón a seguir para TDD, vamos a ver cuál es la mejor forma de llevarlo a cabo.

# Ciclo General

# Escribir una Prueba

- ★ Pensá en cómo te gustaría que apareciera la operación en el código
- ★ Estás escribiendo una historia... inventá la interfaz que desearías tener
- ★ Incluí todos los elementos en la historia que imagines serán necesarios para calcular las respuestas correctas


veamos un ejemplo...

“Quiero diseñar un perro que ladre”...

- `perro.ladrar()`
- `perro.gritar()`
- `perro.emitir_sonido()`
- `perro.guau()`
- `perro.hacer_accion_principal_de_perro()`

¿cuál instrucción pega mejor con nuestra historia?

# ¡Hacer que funcione!

- ★ Conseguir rápidamente la validación del entorno de programación  4/4
- ★ Si una solución limpia y simple es obvia, escribirla
- ★ Si la solución limpia y simple es obvia pero va a llevar más de un minuto, dejá una nota y volvé al problema principal: hacer que la barra se ponga verde en segundos

“Quiero diseñar un perro que coma sólo 1kg de carne cada vez”

```
def test_perro_comer_1kg_carne(self):  
    perro = Perro()  
    self.assertTrue(perro.comer("🍖"))
```

```
def comer(self, comida: str):  
    return True
```

```
✓ 17 def test_perro_comer_1kg_carne(self):  
    18     perro = Perro()  
    19     self.assertTrue(perro.comer("🍖"))
```

```
✗ 20 def test_perro_no_comer_2kg_carne(self):  
    21     perro = Perro()  
    22     self.assertFalse(perro.comer("🍖🍖"))
```

la solución es bastante simple y clara...

queremos que el perro  
sólo coma 1kg de carne...

```
def comer(self, comida: str):  
    if len(comida) > 1: return False  
    return True
```


```
def test_perro_no_come_2kg_carne(self):  
    perro = Perro()  
    self.assertFalse(perro.comer("🍖🍖"))
```

¿esta solución resuelve  
todos los problemas? ¿y  
si le damos de comer otra  
cosa que no sea carne?

```
⊗ 23     def test_perro_no_come_pan(self):           /Us  
    24         perro = Perro()  
    25         self.assertFalse(perro.comer("🍌"))  
    26
```

notemos cómo, cada vez, la solución se va tornando  
más complicada de resolver (y de programar)

# Dejar el código bien escrito

- ★ Una vez que el sistema tiene el comportamiento que corresponde (a.k.a.: tenemos todos los ) , emprolijá el código
- ★ Eliminá la duplicación que se generó para tener las pruebas corriendo más rápido



“Quiero diseñar un perro  
que no coma ni pan, ni sopa,  
ni pescado, ni fideos, ni  
tomate”

una solución rápida:

```
def comer(self, comida: str):  
    if len(comida) > 1: return False  
    if "🍌" in comida: return False  
    if "🍜" in comida: return False  
    if "🍲" in comida: return False  
    if "🍝" in comida: return False  
    if "🍅" in comida: return False  
    return True
```

```
⊗ 26 def test_perro_no_comer_sopa(self):  
27     perro = Perro()  
28     self.assertFalse(perro.comer("🍜"))  
⊗ 29 def test_perro_no_comer_pescado(self):  
30     perro = Perro()  
31     self.assertFalse(perro.comer("🍲"))  
⊗ 32 def test_perro_no_comer_fideos(self):  
33     perro = Perro()  
34     self.assertFalse(perro.comer("🍝"))  
⊗ 35 def test_perro_no_comer_tomate(self):  
36     perro = Perro()  
37     self.assertFalse(perro.comer("🍅"))
```

```
✓ TestPerro  
  ✓ test_perro_ladra  
  ✓ test_perro_pide_comida  
  ✓ test_perro_persigue_gatos  
  ✓ test_perro_corre  
  ✓ test_perro_comer_1kg_carne  
  ✓ test_perro_no_comer_2kg_carne  
  ✓ test_perro_no_comer_pan  
  ✓ test_perro_no_comer_sopa  
  ✓ test_perro_no_comer_pescado  
  ✓ test_perro_no_comer_fideos  
  ✓ test_perro_no_comer_tomate
```

ahora que funciona,  
¿cómo lo emprolijamos?

## un posible refactor...

```
HKEY_COMIDAS_PROHIBIDAS = ["🌮", "🍲", "🍷", "🍝", "🍅"]
```

```
class Perro:
    def ladrar(self): ...

    def pedir_comida(self): ...

    def perseguir_gato(self): ...

    def correr(self): ...

    def comer(self, comida: str):
        if len(comida) > 1: return False
        if comida in HKEY_COMIDAS_PROHIBIDAS: return False
        return True
```

```
✓ TestPerro
  ✓ test_perro_ladra
  ✓ test_perro_pide_comida
  ✓ test_perro_persigue_gatos
  ✓ test_perro_corre
  ✓ test_perro_come_1kg_carne
  ✓ test_perro_no_come_2kg_carne
  ✓ test_perro_no_come_pan
  ✓ test_perro_no_come_sopa
  ✓ test_perro_no_come_pescado
  ✓ test_perro_no_come_fideos
  ✓ test_perro_no_come_tomate
```

# Goal

“The goal is clean code that works. [...] Divide and conquer, baby. First we’ll solve the “that works” part of the problem. Then we’ll solve the ‘clean code’ part.”



# ¿Dudas?



# Convención de Escritura de Pruebas

# Programación Orientada a Objetos (POO)

# Instalación de un Framework de Testing

# Requerimientos

1. Instalar VSCode
2. Instalar Python
3. Instalar la extensión Python Test Explorer for VSCode
4. Reiniciar el entorno (o sea: VSCode)





## Configure Python Tests

Select a test framework/tool to enable

**unittest** Standard Python test framework  
<https://docs.python.org/3/library/unittest.html>

**pytest** pytest framework  
<http://docs.pytest.org/>



Select the directory containing the tests

. Root directory

clase10-tdd



Select the pattern to identify test files

\*test.py Python files ending with 'test'

\*\_test.py Python files ending with '\_test'

test\*.py Python files beginning with 'test'

test\_\*.py Python files beginning with 'test\_'

\*test\*.py Python files containing the word 'test'

# Otros entornos que pueden servir

Si no les convence VSCode, otras opciones son:

- [PyCharm](#) (licencia con correo fi.uba.ar)
- [Spyder](#)

Otros más livianos...

- Eclipse con PyDev
- Sublime Text
- Atom

# ¿break y ejercitamos?



# Ejercitación

# PyCar

Desarrollar una clase “PyCar” que represente un vehículo y pueda realizar distintas acciones, como...

- encender
- apagar
- acelerar
- frenar
- doblar a la izquierda
- doblar a la derecha
- retroceder

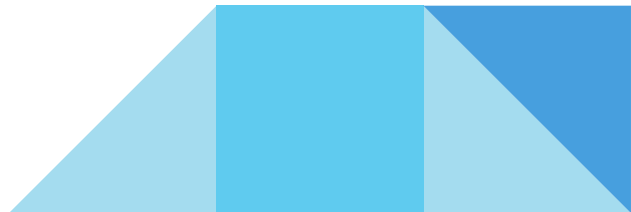
Escribir pruebas que verifiquen que el automóvil se comporta correctamente.

# ¡ahora solos!

# Suma de enteros $\rightarrow 5'$

Escribir una función que reciba dos números enteros por parámetro y devuelva su suma.

Escribir pruebas que definan el comportamiento esperado de la función para diferentes casos de prueba, como números positivos, negativos y cero.





# Validación de contraseña → 15'

Desarrollar una función que valide una contraseña según:

- longitud mínima de 8 caracteres
- inclusión de al menos 1 caracter alfanuméricos
- inclusión de al menos 2 caracteres especiales

Escribir pruebas que verifiquen que la función devuelve resultados correctos para contraseñas válidas e inválidas.

# Para la clase que viene...

- ❑ 2° Parcialito
- ❑ Depuración (debugging)
- ❑ Ejercitación: Debugging +  
TDD

