

# **Coverage, Black-Box Testing y White-Box Testing**

**IIC3745 - Testing**

**Maximiliano Narea Carvajal**

# CGI Decoder

## Coverage

```
1 def cgi_decode(s: str) -> str:
2     """Decode the CGI-encoded string `s`:
3         * replace '+' by ' '
4         * replace "%xx" by the character with hex number xx.
5         Return the decoded string. Raise `ValueError` for invalid inputs."""
6
7     # Mapping of hex digits to their integer values
8     hex_values = {
9         '0': 0, '1': 1, '2': 2, '3': 3, '4': 4,
10        '5': 5, '6': 6, '7': 7, '8': 8, '9': 9,
11        'a': 10, 'b': 11, 'c': 12, 'd': 13, 'e': 14, 'f': 15,
12        'A': 10, 'B': 11, 'C': 12, 'D': 13, 'E': 14, 'F': 15,
13    }
14
15    t = ""
16    i = 0
17    while i < len(s):
18        c = s[i]
19        if c == '+':
20            t += ' '
21        elif c == '%':
22            digit_high, digit_low = s[i + 1], s[i + 2]
23            i += 2
24            if digit_high in hex_values and digit_low in hex_values:
25                v = hex_values[digit_high] * 16 + hex_values[digit_low]
26                t += chr(v)
27            else:
28                raise ValueError("Invalid encoding")
29        else:
30            t += c
31        i += 1
32    return t
```

# ¿Cómo funciona?

## Coverage

```
[4] 1 cgi_decode("Hello+World")
```

```
⇒ 'Hello World'
```

**¿Cómo podemos testear esta  
función?**

# **Black-Box Testing & White-Box Testing**

# Black-Box Testing

# ¿En qué consiste?

## Black-Box Testing

La idea del Black Box Testing es crear pruebas a partir de las especificaciones. En el caso anterior, tendríamos que probar `cgi_decode()` mediante las características especificadas y documentadas, incluyendo:

- Prueba de reemplazo correcto de '+'.
- Prueba de reemplazo correcto de "%xx".
- Prueba de no reemplazo de otros caracteres.
- Prueba de reconocimiento de entradas ilegales.

# Testeamos comportamientos esperados

## Black-Box Testing

```
1 assert cgi_decode('+') == ' '  
2 assert cgi_decode('%20') == ' '  
3 assert cgi_decode('abc') == 'abc'  
4  
5 try:  
6     cgi_decode('%?a')  
7     assert False  
8 except ValueError:  
9     pass
```



# White-Box Testing

# ¿En qué consiste?

## White-Box Testing

El testing de caja blanca se basa en la **estructura interna del código**, asegurando que todas las declaraciones se ejecuten durante las pruebas. Esto implica **cumplir con criterios de cobertura** específicos para **garantizar la suficiencia de las pruebas**, siendo estos criterios fundamentales para detectar posibles errores en el código.

# **Recordatorio de la clase pasada**

# ¿En qué consiste el Coverage?

## Coverage

- Es una medida que nos permite determinar qué tanto código hemos probado.
  - Existen diversas métricas, pero hoy vamos a revisar solo dos.
  - En general, se apunta a tener un porcentaje de coverage alto, ya que indica que hemos revisado el comportamiento de ciertas partes del código.
- (disclaimer)

# Statement Coverage

## Coverage

Probamos la mayor cantidad de líneas de código

En este ejemplo, basta con testear `open_resources()`




```
1 class Wallet:
2     def __init__(self, id):
3         self.id = id
4         self.balance = 0
5         self.resources_open = False
6
7     def deposit(self, amount):
8         self.balance += amount
9
10    def withdraw(self, amount):
11        self.balance -= amount
12
13    def total_balance(self):
14        return self.balance
15
16    def open_resources(self):
17        self.resources_open = True
18
19    def close_resources(self):
20        self.resources_open = False
```

# Branch Coverage

## Coverage

Probamos todas las condiciones posibles.

En este ejemplo se requieren por lo menos dos tests distintos



```
1 def test_function(x, y):  
2     if x > y:  
3         return true  
4     else:  
5         return false
```

**¿Cómo testeamos con  
White-Box Testing?**

```
1 def cgi_decode(s: str) -> str:
2     t = ""
3     i = 0
4     while i < len(s):
5         c = s[i]
6         if c == '+':
7             t += ' '
8         elif c == '%':
9             digit_high, digit_low = s[i + 1], s[i + 2]
10            i += 2
11            if digit_high in hex_values and digit_low in hex_values:
12                v = hex_values[digit_high] * 16 + hex_values[digit_low]
13                t += chr(v)
14            else:
15                raise ValueError("Invalid encoding")
16        else:
17            t += c
18            i += 1
19    return t
```

```
1 assert cgi_decode('+') == ' '
2 assert cgi_decode('%20') == ' '
3 assert cgi_decode('abc') == 'abc'
```



```
1 def cgi_decode(s: str) -> str:
2     t = ""
3     i = 0
4     while i < len(s):
5         c = s[i]
6         if c == '+':
7             t += ' '
8         elif c == '%':
9             digit_high, digit_low = s[i + 1], s[i + 2]
10            i += 2
11            if digit_high in hex_values and digit_low in hex_values:
12                v = hex_values[digit_high] * 16 + hex_values[digit_low]
13                t += chr(v)
14            else:
15                raise ValueError("Invalid encoding")
16        else:
17            t += c
18            i += 1
19    return t
```

```
1 assert cgi_decode('+') == ' '
2 assert cgi_decode('%20') == ' '
3 assert cgi_decode('abc') == 'abc'
```

```
1 def cgi_decode(s: str) -> str:
2     t = ""
3     i = 0
4     while i < len(s):
5         c = s[i]
6         if c == '+':
7             t += ' '
8         elif c == '%':
9             digit_high, digit_low = s[i + 1], s[i + 2]
10            i += 2
11            if digit_high in hex_values and digit_low in hex_values:
12                v = hex_values[digit_high] * 16 + hex_values[digit_low]
13                t += chr(v)
14            else:
15                raise ValueError("Invalid encoding")
16        else:
17            t += c
18            i += 1
19    return t
```

```
1 assert cgi_decode('+') == ' '
2 assert cgi_decode('%20') == ' '
3 assert cgi_decode('abc') == 'abc'
```

```
1 def cgi_decode(s: str) -> str:
2     t = ""
3     i = 0
4     while i < len(s):
5         c = s[i]
6         if c == '+':
7             t += ' '
8         elif c == '%':
9             digit_high, digit_low = s[i + 1], s[i + 2]
10            i += 2
11            if digit_high in hex_values and digit_low in hex_values:
12                v = hex_values[digit_high] * 16 + hex_values[digit_low]
13                t += chr(v)
14            else:
15                raise ValueError("Invalid encoding")
16        else:
17            t += c
18            i += 1
19    return t
```

```
1 assert cgi_decode('+') == ' '
2 assert cgi_decode('%20') == ' '
3 assert cgi_decode('abc') == 'abc'
```

# En resumen...

## White-Box Testing

- Al igual que en las pruebas de caja negra, cubrir todas las alternativas suele garantizar la cobertura de los diferentes comportamientos especificados, ya que **los programadores implementan comportamientos distintos en diferentes partes del código**. Esto resulta en **una correspondencia común** entre la cobertura del código y los casos de prueba.
- Las pruebas de caja blanca encuentran errores en el comportamiento implementado y ayuda a identificar casos extremos que no están bien especificados. Sin embargo, **puede no detectar funcionalidades que faltan en la implementación**, ya que se enfoca en el código existente.

**¿Cómo se obtiene el  
coverage en la práctica?**

# Tracing Executions

## ¿Cómo se obtiene el coverage en la práctica?

En general obtener el coverage es una **tarea difícil**, pero Python tiene algunos atajos que nos simplifican la tarea.

# Tracing Executions

## ¿Cómo se obtiene el coverage en la práctica?

- Para esto utilizaremos la funcionalidad *sys.settrace()*
- **frame:** permite acceder al marco actual, permitiendo el acceso a la ubicación y las variables actuales:
  - **frame.f\_code** es el código que se está ejecutando actualmente con **frame.f\_code.co\_name** siendo el nombre de la función;
  - **frame.f\_lineno** contiene el número de línea actual.
  - **frame.f\_locals** contiene las variables locales y argumentos actuales.

# Tracing Executions

## ¿Cómo se obtiene el coverage en la práctica?

- **event:** es una cadena con valores que incluyen "line" (se ha alcanzado una nueva línea) o "call" (se está llamando a una función).
- **arg:** es un argumento adicional para algunos eventos; para los eventos "return", por ejemplo, arg contiene el valor que se está devolviendo.





```
1 import sys
2 from types import FrameType, TracebackType
3
4 coverage = []
5
6 def traceit(frame, event, arg):
7     """Trace program execution. To be passed to sys.settrace()."""
8     if event == 'line':
9         global coverage
10         function_name = frame.f_code.co_name
11         lineno = frame.f_lineno
12         coverage.append(lineno)
13     return traceit
14
15 def cgi_decode_traced(s):
16     global coverage
17     coverage = []
18     sys.settrace(traceit) # Turn on
19     cgi_decode(s)
20     sys.settrace(None)    # Turn off
21
```

```
1 cgi_decode_traced( "a+b" )
2 print(coverage)
```

```
1 [8, 9, 8, 9, 8, 9, 8, 9, 8, 9, 8, 10, 8, 10, 8, 10, 8, 10, 8, 10,
  8, 11, 8, 11, 8, 11, 8, 11, 8, 11, 8, 11, 8, 12, 8, 12, 8, 15, 16,
  17, 18, 19, 21, 30, 31, 17, 18, 19, 20, 31, 17, 18, 19, 21, 30, 31,
  17, 32]
```

```
1 covered_lines = set(coverage)
2 print(covered_lines)
```

```
1 {32, 8, 9, 10, 11, 12, 15, 16, 17, 18, 19, 20, 21, 30, 31}
```

```
1 class Coverage:
2     """Track coverage within a `with` block. Use as
3     ```
4     with Coverage() as cov:
5         function_to_be_traced()
6     c = cov.coverage()
7     ```
8     """
9
10    def __init__(self) -> None:
11        """Constructor"""
12        self._trace: List[Location] = []
13
14    # Trace function
15    def traceit(self, frame: FrameType, event: str, arg: Any) -> Optional[Callable]:
16        """Tracing function. To be overloaded in subclasses."""
17        if self.original_trace_function is not None:
18            self.original_trace_function(frame, event, arg)
19
20        if event == "line":
21            function_name = frame.f_code.co_name
22            lineno = frame.f_lineno
23            if function_name != '__exit__': # avoid tracing ourselves:
24                self._trace.append((function_name, lineno))
25
26        """The set of executed lines, as (function_name, line_number) pairs"""
27        return set(self.trace())
28
29    ....
```

# ¿Cuál es la gracia?

¿Cómo se obtiene el coverage en la práctica?

```
1 with Coverage() as cov_plus:
2     cgi_decode("a+b")
3 with Coverage() as cov_standard:
4     cgi_decode("abc")
5
6 cov_plus.coverage() -
  cov_standard.coverage()
```

```
1 with Coverage() as cov_max:
2     cgi_decode('+')
3     cgi_decode('%20')
4     cgi_decode('abc')
5     try:
6         cgi_decode('%?a')
7     except Exception:
8         pass
```

# Visibilidad por cada test

¿Cómo se obtiene el coverage en la práctica?

```
1 cov_max.coverage() - cov_plus.coverage()
```

```
1 {('cgi_decode', 22), ('cgi_decode', 23), ('cgi_decode', 24),  
   ('cgi_decode', 25), ('cgi_decode', 26), ('cgi_decode', 28)}
```

# Fuzzing Básico

# ¿Qué es un Fuzzer?

## Fuzzing Básico

```
1 from Fuzzer import fuzzer
2 sample = fuzzer()
3 sample
4
5 ' !7#%"*#0=)$;%6*; >638:*>80"= </> ( /*: -( 2<4 !:5*6856&?" "11<7+%
  <%7,4.8,*+&,,$,." '
6
```

# ¿Y de qué sirve en este contexto?

## Fuzzing Básico

```
1 with Coverage() as cov_fuzz:
2     try:
3         cgi_decode(sample)
4     except:
5         pass
6 cov_max.coverage() - cov_fuzz.coverage()
7
8 {('cgi_decode', 20), ('cgi_decode', 25),
9  ('cgi_decode', 26), ('cgi_decode', 32)}
```



# ¿Cómo lo usamos?

## Fuzzing Básico

```
1 trials = 100
2 def population_coverage(population: List[str], function:
  Callable):
3     cumulative_coverage: List[int] = []
4     all_coverage: Set[Location] = set()
5
6     for s in population:
7         with Coverage() as cov:
8             try:
9                 function(s)
10            except:
11                pass
12            all_coverage |= cov.coverage()
13            cumulative_coverage.append(len(all_coverage))
14
15     return all_coverage, cumulative_coverage
16
```

# ¿Y de qué sirve en este contexto?

## Fuzzing Básico

```
1 all_coverage, cumulative_coverage = population_coverage(hundred_inputs(), cgi_decode)
2 plt.plot(cumulative_coverage)
3 plt.title('Coverage of cgi_decode() with random inputs')
4 plt.xlabel('# of inputs')
5 plt.ylabel('lines covered')
```

Coverage of cgi\_decode() with random inputs

