

LLMs en Programación

De la generación de código a la solución de problemas

Clase 4: Solución de Proble-
mas de Programación con LLMs

Christian Luna Escudero

Grupo de investigación KDIS
Instituto de investigación DaSCI.
Universidad de Córdoba

25 de junio de 2025

KDIS Lab



DaSCI

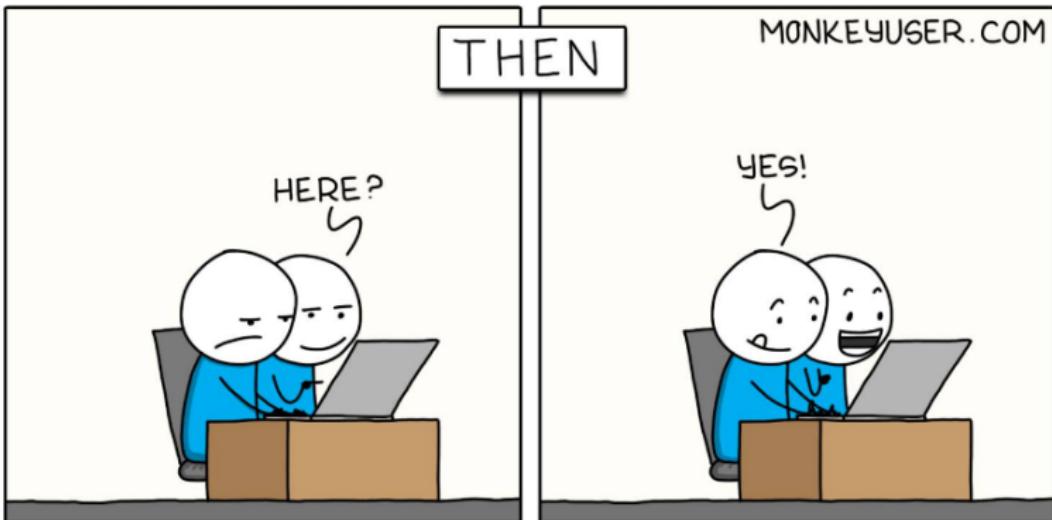
¿Qué aprenderéis hoy?

Cómo usar modelos de lenguaje como copilotos inteligentes:

- Para entender código heredado o mal documentado.
- Para proponer refactorizaciones sin romper funcionalidades.
- Para generar tests automáticos útiles.
- Para documentar código técnico con estándares claros.

¿Pair Programming antes?

PAIR PROGRAMMING : THEN vs. NOW



¿Pair Programming ahora?



¿Pair programming en todo momento?

- Los LLMs pueden actuar como copilotos inteligentes, pero ¿deberíamos usarlos siempre?
- ¿Qué tareas se benefician más del acompañamiento continuo de un modelo?
- ¿Dónde puede entorpecer o inducir errores?
- ¿Cómo equilibrar eficiencia, aprendizaje personal y supervisión?
- ¿Qué papel juega el juicio humano en esta colaboración?

¿Dónde trazamos la línea entre asistencia útil y dependencia excesiva?

¿Pair programming en todo momento?

- Los LLMs pueden actuar como copilotos inteligentes, pero ¿deberíamos usarlos siempre?
- ¿Qué tareas se benefician más del acompañamiento continuo de un modelo?
- ¿Dónde puede entorpecer o inducir errores?
- ¿Cómo equilibrar eficiencia, aprendizaje personal y supervisión?
- ¿Qué papel juega el juicio humano en esta colaboración?

¿Dónde trazamos la línea entre asistencia útil y dependencia excesiva?

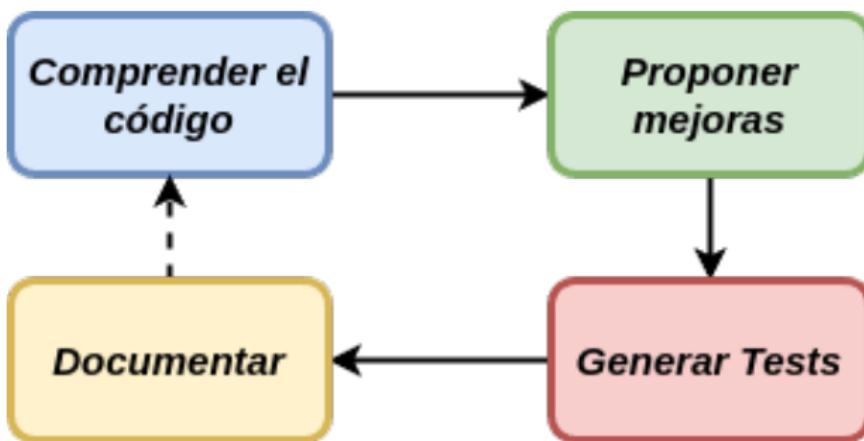
Evolución de la ayuda en la programación



Caso central: función a analizar

```
1  def proc_data(d):
2      r = []
3      for i in range(len(d)):
4          if isinstance(d[i], int):
5              if d[i] % 2 == 0:
6                  r.append(d[i] * 2)
7              else:
8                  r.append(d[i] + 1)
9              elif isinstance(d[i], str) and d[i].isdigit():
10                  r.append(int(d[i]) + 100)
11              else:
12                  continue
13      return sorted(r, reverse=True)
14
```

Estrategia general



Comprendión asistida

Prompt:

Actúa como un ingeniero de software senior. Analiza esta función de Python y explica en lenguaje técnico qué hace, qué inputs acepta y qué salida produce.

```
1  def proc_data(d):
2      r = []
3      for i in range(len(d)):
4          if isinstance(d[i], int):
5              if d[i] % 2 == 0:
6                  r.append(d[i] * 2)
7              else:
8                  r.append(d[i] + 1)
9              elif isinstance(d[i], str) and d[i].isdigit():
10                  r.append(int(d[i]) + 100)
11              else:
12                  continue
13      return sorted(r, reverse=True)
14
```

¿Qué es refactorizar?

- Proceso de modificar el código sin alterar su comportamiento externo.
- Mejora su legibilidad, estructura o rendimiento.
- Facilita el mantenimiento, extensibilidad y detección de errores.

¿Cuándo conviene refactorizar?

- Código difícil de entender o duplicado.
- Cambio de requisitos o incorporación de nuevas funcionalidades.
- Mejoras en eficiencia, claridad o estilo de codificación.
- Antes de escribir pruebas unitarias.

Tipos comunes de refactorización (I)

- Renombrado de variables y funciones (nombres más descriptivos).
- Extracción de funciones: dividir en unidades más manejables.
- Eliminación de código duplicado.

Tipos comunes de refactorización (II)

- Simplificación de estructuras de control (if, switch).
- Sustitución de código complejo por funciones auxiliares.
- Separación de responsabilidades (Single Responsibility Principle).

¿Cómo ayudan los LLMs a refactorizar? (I)

- Sugerencias de renombrado más semántico y consistente.
- Detección de funciones muy largas y recomendaciones de extracción.
- Identificación de duplicación de código en distintos archivos.

¿Cómo ayudan los LLMs a refactorizar? (II)

- Reorganización de clases y funciones siguiendo principios SOLID.
- Propuesta de estructuras más eficientes o idiomáticas.
- Reformulación de bucles y condicionales innecesariamente complejos.

Limitaciones de la refactorización con LLMs

- No garantizan que el comportamiento original se conserve.
- A veces eliminan partes relevantes del código sin intención.
- Se requiere siempre validación manual y testing.

Caso Práctico: Refactorización de código

Prompt Base:

Actúa como un ingeniero de software senior. Refactoriza la siguiente función de Python para que sea más clara y siga principios de diseño (como SRP y legibilidad). Preserva exactamente su comportamiento.

Código: <... código anterior ...>

Caso Práctico 1: Refactorización de código con Github Copilot

Código Original

```
1  def proc_data(d):
2      r = []
3      for i in range(len(d)):
4          if isinstance(d[i], int):
5              if d[i] % 2 == 0:
6                  r.append(d[i] * 2)
7              else:
8                  r.append(d[i] + 1)
9          elif isinstance(d[i], str) and d[i].isdigit():
10              r.append(int(d[i]) + 100)
11          else:
12              continue
13      return sorted(r, reverse=True)
14
```

Caso Práctico 1: Refactorización de código con Github Copilot

Código Original

```
1 def proc_data(d):
2     r = []
3     for i in range(len(d)):
4         if isinstance(d[i], int):
5             if d[i] % 2 == 0:
6                 r.append(d[i] * 2)
7             else:
8                 r.append(d[i] + 1)
9         elif isinstance(d[i], str) and d[i].isdigit():
10            r.append(int(d[i]) + 100)
11        else:
12            continue
13    return sorted(r, reverse=True)
14
```

Código Propuesto por Copilot

```
def process_integer(value):
    """Process integer: double if even, increment by 1 if odd."""
    return value * 2 if value % 2 == 0 else value + 1

def process_digit_string(value):
    """Convert digit string to int and add 100."""
    return int(value) + 100

def proc_data(data):
    """Process a list of mixed types, applying rules to ints and digit strings."""
    results = []
    for item in data:
        if isinstance(item, int):
            results.append(process_integer(item))
        elif isinstance(item, str) and item.isdigit():
            results.append(process_digit_string(item))
        # Ignore other types
    return sorted(results, reverse=True)
```

Caso Práctico 2: Refactorización de código con ChatGPT

Código Original

```
1  def proc_data(d):
2      r = []
3      for i in range(len(d)):
4          if isinstance(d[i], int):
5              if d[i] % 2 == 0:
6                  r.append(d[i] * 2)
7              else:
8                  r.append(d[i] + 1)
9          elif isinstance(d[i], str) and d[i].isdigit():
10             r.append(int(d[i]) + 100)
11         else:
12             continue
13     return sorted(r, reverse=True)
14
```

Caso Práctico 2: Refactorización de código con ChatGPT

Código Original

```
1 def proc_data(d):
2     r = []
3     for i in range(len(d)):
4         if isinstance(d[i], int):
5             if d[i] % 2 == 0:
6                 r.append(d[i] * 2)
7             else:
8                 r.append(d[i] + 1)
9         elif isinstance(d[i], str) and d[i].isdigit():
10            r.append(int(d[i]) + 100)
11        else:
12            continue
13    return sorted(r, reverse=True)
14
```

Código Propuesto por ChatGPT

Claro, vamos a refactorizar la función siguiendo principios como:

- SRP (Single Responsibility Principle): separamos cada tipo de procesamiento en funciones específicas.
- Legibilidad y nombrado claro.
- Evitar acceso directo por índice si no es necesario.
- Evitar estructuras innecesariamente anidadas.

Aquí está la versión refactorizada:

```
python ⌂ Copiar ⌂ Editar

def process_integer(n):
    """Procesa un entero: si es par, lo duplica; si es impar, le suma 1."""
    return n * 2 if n % 2 == 0 else n + 1

def process_string(s):
    """Convierte una cadena de dígitos en entero y le suma 100."""
    return int(s) + 100

def is_digit_string(s):
    """Verifica si una cadena representa un número entero."""
    return s.isdigit()

def proc_data(data):
    """Procesa una lista mixta de enteros y strings según reglas específicas."""
    result = []
    for item in data:
        if isinstance(item, int):
            result.append(process_integer(item))
        elif is_digit_string(item):
            result.append(process_string(item))
        # Si no cumple ninguna condición, se ignora
    return sorted(result, reverse=True)
```

Ventajas de esta refactorización:

- El comportamiento original se mantiene exactamente.

Caso Práctico 3: Refactorización de código con LM Studio

Código Original

```
1  def proc_data(d):
2      r = []
3      for i in range(len(d)):
4          if isinstance(d[i], int):
5              if d[i] % 2 == 0:
6                  r.append(d[i] * 2)
7              else:
8                  r.append(d[i] + 1)
9          elif isinstance(d[i], str) and d[i].isdigit():
10             r.append(int(d[i]) + 100)
11         else:
12             continue
13     return sorted(r, reverse=True)
14
```

Caso Práctico 3: Refactorización de código con LM Studio

Código Original

```
1 def proc_data(d):
2     r = []
3     for i in range(len(d)):
4         if isinstance(d[i], int):
5             if d[i] % 2 == 0:
6                 r.append(d[i] * 2)
7             else:
8                 r.append(d[i] + 1)
9         elif isinstance(d[i], str) and d[i].isdigit():
10            r.append(int(d[i]) + 100)
11        else:
12            continue
13    return sorted(r, reverse=True)
```

Código Propuesto por LM Studio

prueba/prueba-3-123

De acuerdo. Aquí tienes una versión refactorizada de la función `proc_data`, junto con explicaciones sobre los cambios realizados y por qué se hicieron.

```
def process_data(data):
    """Procesa una lista de datos, transformando enteros pares y cadenas numéricas.

    Args:
        data (list): Una lista de elementos que pueden ser enteras o cadenas.

    Returns:
        list: Una nueva lista con los elementos transformados, ordenada de forma descendente.

    """
    transformed_data = []
    for item in data:
        if isinstance(item, int):
            transformed_data.append(_process_integer(item))
        elif isinstance(item, str) and item.isdigit():
            transformed_data.append(_process_string(item))

    return sorted(transformed_data, reverse=True)

def _process_integer(number):
    """Procesa un número entero. Si es par, lo multiplica por 2; si no, lo suma 1."""
    if number % 2 == 0:
        return number * 2
    else:
        return number + 1

def _process_string(string_number):
    """Procesa una cadena que representa un número. La suma 100."""
    return int(string_number) + 100
```

Comentarios y Justificación:

1. **Responsabilidad Única (SRP):** La función original hacía dos cosas: transformar los datos y ordenar los resultados. La refactorización separa estas responsabilidades en dos funciones:
 - `process_data`: Recibe la lista de datos, aplica transformaciones a cada elemento y devuelve una lista transformada.
 - `_process_data`: no ordena los datos. La ordenación se realiza justo antes de devolver el resultado, lo que mantiene la función principal más eficiente.
2. **Funciones Auxiliares (Extracción de Método):** La lógica para procesar enteras y cadenas se extrae en funciones auxiliares privadas (`_process_integer` y `_process_string`). Esto mejora la legibilidad al reducir la complejidad dentro de la función principal. El prefijo `_` indica que son funciones internas (no accesibles por el cliente).

Type a message and press Enter to send...

User (AI) Insert (UI) Clear

Consejo: 0.0% tipo

Discusión del caso práctico

- ¿Qué ventajas tiene la versión propuesta?
- ¿Mantiene exactamente el mismo comportamiento?
- ¿Hay algún *edge case* que se podría pasar por alto?

¿Qué son las pruebas unitarias?

- Son tests que verifican el correcto funcionamiento de unidades pequeñas de código (funciones o métodos).
- Ayudan a detectar errores de forma temprana y garantizan que los cambios no rompan funcionalidades existentes.
- Herramientas comunes: `unittest`, `pytest`, `Jest`, etc.

¿Por qué usar LLMs para pruebas?

- Automatizan la escritura de tests a partir del código fuente o su descripción.
- Pueden proponer *edge cases* o raros que el programador no haya considerado.
- Aceleran el proceso de creación de tests cuando el código es extenso o repetitivo.

Caso Práctico: Generación de Test

Prompt Base:

Escribe pruebas unitarias usando pytest para la siguiente función. Asegúrate de cubrir casos típicos, valores límite y entradas no válidas. Código: <... código anterior ...>

Caso Práctico 1: Generación de Test con Github Copilot

Código Original

```
1  def proc_data(d):
2      r = []
3      for i in range(len(d)):
4          if isinstance(d[i], int):
5              if d[i] % 2 == 0:
6                  r.append(d[i] * 2)
7              else:
8                  r.append(d[i] + 1)
9          elif isinstance(d[i], str) and d[i].isdigit():
10             r.append(int(d[i]) + 100)
11         else:
12             continue
13     return sorted(r, reverse=True)
14
```

Caso Práctico 1: Generación de Test con Github Copilot

Código Original

```
1 def proc_data(d):
2     r = []
3     for i in range(len(d)):
4         if isinstance(d[i], int):
5             if d[i] % 2 == 0:
6                 r.append(d[i] * 2)
7             else:
8                 r.append(d[i] + 1)
9         elif isinstance(d[i], str) and d[i].isdigit():
10            r.append(int(d[i]) + 100)
11        else:
12            continue
13    return sorted(r, reverse=True)
```

Test propuestos por Github Copilot

```
def test_proc_data_typical():
    assert proc_data([2, 3, 4, 'abc', 7]) == [102, 8, 4]
    # 2*2=4, 3+1=4, 4*2=8, abc ignored, 7+100=107; sorted: [102, 8, 4, 4] (but only one 4 from int, one from str)
    assert proc_data([1, 2, 3, 4]) == [8, 4, 2]
    # 1+1=2, 2*2=4, 3+1=4, 4*2=8; sorted: [8, 4, 4, 2]

def test_proc_data_with_strings():
    assert proc_data(['10', '5', '4', '0']) == [110, 105, 100]
    # 10*2=200, 5+105, 4+100, 0+100

def test_proc_data_with_invalid_types():
    assert proc_data([None, (), {}]) == []
    # Only 0 is valid: 2*0=0

def test_proc_data_empty():
    assert proc_data([]) == []

def test_proc_data_all_invalid():
    assert proc_data(['a', (), None, ()]) == []

def test_proc_data_large_numbers():
    assert proc_data([1000000, 999999]) == [2000000, 1000000]
    # 1000000*2>2000000, 999999<-1000000

def test_proc_data_negative_numbers():
    assert proc_data([-2, -3, '-4', '-5']) == [-1, -1]
    # -2*2=-4, -3+1=-2, '-4' and '-5' are not digit strings, so ignored; sorted: [-1, -1]

def test_proc_data_mixed_types():
    assert proc_data([2, 3, 4.5, True, False]) == [102, 4]
    # 2*2=4, 3+1=4, 4.5 ignored, True=False are bool (subclass of int), True=1>0, False=0>0; sorted: [102, 4, 2, 0]
    # Not isinstance(True, int) is True, so True=1>2, False=0>0

def test_proc_data_duplicated():
    assert proc_data([2, 2, '2']) == [104, 4, 4]
    # 2*2=4, 2+2=4, '2' ignored; sorted: [102, 102, 4, 4]

def test_proc_data_only_strings():
    assert proc_data(['1', '2', '3']) == [103, 102, 101]
    # 1+1=2, 2+2=4, 3+3=6; sorted: [103, 102, 101]
```

Caso Práctico 2: Generación de Test con ChatGPT

Código Original

```
1  def proc_data(d):
2      r = []
3      for i in range(len(d)):
4          if isinstance(d[i], int):
5              if d[i] % 2 == 0:
6                  r.append(d[i] * 2)
7              else:
8                  r.append(d[i] + 1)
9          elif isinstance(d[i], str) and d[i].isdigit():
10             r.append(int(d[i]) + 100)
11         else:
12             continue
13     return sorted(r, reverse=True)
14
```

Caso Práctico 2: Generación de Test con ChatGPT

Código Original

```
1 def proc_data(d):
2     r = []
3     for i in range(len(d)):
4         if isinstance(d[i], int):
5             if d[i] % 2 == 0:
6                 r.append(d[i] * 2)
7             else:
8                 r.append(d[i] + 1)
9         elif isinstance(d[i], str) and d[i].isdigit():
10            r.append(int(d[i]) + 100)
11        else:
12            continue
13    return sorted(r, reverse=True)
```

Test propuestos por ChatGPT

Perfecto. A continuación te dejo un conjunto de **pruebas unitarias** usando `pytest` que cubren:

- **Casos típicos** (enteros pares/impares, strings numéricos),
- **Valores límite** (números grandes, cero, strings vacíos),
- **Entradas no válidas** (None, listas, objetos mixtos, strings no numéricos, etc.).

Guarda esto en un archivo como `test_proc_data.py`:

```
python ⌂ Copiar ⌂ Editar
import pytest
from your_module import proc_data # Cambia 'your_module' por el nombre real del módulo si es necesario

def test_all_even_integers():
    assert proc_data([2, 4, 6]) == [12, 8, 4]

def test_all_odd_integers():
    assert proc_data([1, 3, 5]) == [6, 4, 2]

def test_mixed_integers():
    assert proc_data([1, 2, 3, 4]) == [8, 6, 4, 2]

def test_string_digits():
    assert proc_data(["1", "2", "00"]) == [100, 102, 101]

def test_mixed_types():
    assert proc_data([1, "2", "abc", 4, None, [5, 5], "00"]) == [100, 8, 4, 102]

def test_empty_list():
    assert proc_data([]) == []

def test_all_invalid_types():
    assert proc_data([None, (), {}, "abc", 3.14]) == []

def test_zero_and_edge_cases():
    assert proc_data([0, "0"]) == [100, 0] # 0*2 = 0; "0" + 100 = 100

def test_negative_numbers():
```

Caso Práctico 3: Generación de Test con LM Studio

Código Original

```
1  def proc_data(d):
2      r = []
3      for i in range(len(d)):
4          if isinstance(d[i], int):
5              if d[i] % 2 == 0:
6                  r.append(d[i] * 2)
7              else:
8                  r.append(d[i] + 1)
9          elif isinstance(d[i], str) and d[i].isdigit():
10             r.append(int(d[i]) + 100)
11         else:
12             continue
13     return sorted(r, reverse=True)
14
```

Caso Práctico 3: Generación de Test con LM Studio

Test propuestos por LM Studio

```

import pytest

def test_pro_data():
    r = []
    for i in range(len(d1)):
        if isinstance(d1[i], dict):
            if i % 2 == 0:
                r.append(d1[i] + 2)
            else:
                r.append(d1[i] + 1)
        elif isinstance(d1[i], str) and d1[i].isdigit():
            r.append(int(d1[i]) * 100)
        else:
            continue
    return sorted(r, reverse=True)

def test_pro_data_empty_list():
    assert pro_data() == []

def test_pro_data_only_one_number():
    assert pro_data([2, 4, 6]) == [6, 8, 4]

def test_pro_data_only_odd_numbers():
    assert pro_data([1, 3, 5, 7]) == [6, 4, 2]

def test_pro_data_mixed_numbers():
    assert pro_data([2, 4, 6, 8, 10]) == [10, 6, 4, 2, 8]

def test_pro_data_with_string():
    assert pro_data(["a", "b", "c", "d"]) == [10, 12, 8, 6]

def test_pro_data_with_no_digits_string():
    assert pro_data(["abc", "11"]) == [8, 4, 2]

def test_pro_data_with_true():
    assert pro_data([1, 2, 3, 4]) == [6, 4, 2]

def test_pro_data_with_None():
    assert pro_data([None, None]) == [6, 4, 2]

def test_pro_data_with_no_boolean():
    assert pro_data([True, None]) == [6, 4, 2]

def test_pro_data_with_mixed_types():
    assert pro_data([1, 2, 3, True, "a", None]) == [102, 100, 98, 96, 94, 92]

def test_pro_data_large_number():
    assert pro_data([1000, 1000]) == [1000, 1000]

def test_pro_data_repeating_number():
    assert pro_data([-1, -2]) == [6, 4]

def test_pro_data_an_error():
    assert pro_data([0]) == [4]

def test_pro_data_duplicate_values():
    assert pro_data([4]) == [4]

def test_pro_data_duplicate_values():

```

Evaluación de la cobertura

- ¿Se cubren todos los caminos posibles de ejecución?
- ¿Se consideran inputs inesperados o vacíos?
- ¿El LLM propone asserts claros y con buen criterio?

Limitaciones al usar LLMs para testing

- A veces no detecta errores lógicos sutiles.
- Puede asumir el comportamiento esperado sin pruebas empíricas.
- Es imprescindible revisar manualmente la cobertura y la relevancia.

¿Qué es documentación técnica?

- Explicaciones del propósito, uso y estructura del código.
- Incluye: docstrings, comentarios, guías de uso, README, etc.
- Es esencial para el mantenimiento, colaboración y reutilización.

Tipos comunes de documentación

- **Docstrings:** explican funciones, clases y módulos.
- **Comentarios en línea:** explicaciones breves de lógica compleja.
- **Guías de uso:** README, ejemplos y tutoriales.
- **Documentación generada:** herramientas como Sphinx o MkDocs.

Herramientas de documentación en Python

Al igual que en otros lenguajes de programación existe una gran variedad de herramientas que buscan agilizar y elaborar documentos técnicos que permitan comprender mejor el código.

- Sphinx
- pydoctor
- pydoc
- Doxygen

¿Cómo ayudan los LLMs?

- Generan docstrings automáticos a partir del código fuente.
- Crean descripciones de clases y estructuras de forma clara.
- Generan ejemplos de uso y explicaciones para usuarios no expertos.

Estilos populares de documentación

- **Google style:** fácil de leer, usado por TensorFlow, etc.
- **NumPy style:** común en ciencia de datos.
- **reStructuredText:** integrable con Sphinx.

Caso Práctico: Agilizar la documentación

Prompt Base:

Genera un docstring para la siguiente función usando el estilo NumPy. Incluye descripción general, parámetros, retornos y ejemplos.

Código: <... código anterior ...>

Caso Práctico 1: Agilizar la documentación con Github Copilot

Código Original

```
1 def proc_data(d):
2     r = []
3     for i in range(len(d)):
4         if isinstance(d[i], int):
5             if d[i] % 2 == 0:
6                 r.append(d[i] * 2)
7             else:
8                 r.append(d[i] + 1)
9         elif isinstance(d[i], str) and d[i].isdigit():
10            r.append(int(d[i]) + 100)
11        else:
12            continue
13    return sorted(r, reverse=True)
14
```

Caso Práctico 1: Agilizar la documentación con Github Copilot

Código Original

```
1 def proc_data(d):
2     r = []
3     for i in range(len(d)):
4         if isinstance(d[i], int):
5             if d[i] % 2 == 0:
6                 r.append(d[i] * 2)
7             else:
8                 r.append(d[i] + 1)
9         elif isinstance(d[i], str) and d[i].isdigit():
10            r.append(int(d[i]) + 100)
11        else:
12            continue
13    return sorted(r, reverse=True)
14
```

Respuesta de Github Copilot

```
def proc_data(d):
    """
    Processes a list of elements by applying specific transformations to integers and numeric strings.

    For each element in the input list:
    - If the element is an integer:
        - If it is even, multiply it by 2.
        - If it is odd, add 1 to it.
    - If the element is a string representing a digit, convert it to an integer and add 100.
    - Other types are ignored.

    The resulting list is sorted in descending order before being returned.

    Parameters
    ---------
    d : list
        List of elements to process. Elements can be integers, strings, or other types.

    Returns
    -------
    list
        A list of processed integers, sorted in descending order.

    Examples
    ---------
    >>> proc_data([2, 3, '4', 'abc', ?])
    [104, 4, 4, 8]
    >>> proc_data(['10', 5, 6, 'xyz'])
    [102, 12, 6]
    >>>
    r = []
    for i in range(len(d)):
        if isinstance(d[i], int):
            if d[i] % 2 == 0:
                r.append(d[i] * 2)
            else:
                r.append(d[i] + 1)
        elif isinstance(d[i], str) and d[i].isdigit():
            r.append(int(d[i]) + 100)
        else:
            continue
    return sorted(r, reverse=True)
```

Caso Práctico 2: Agilizar la documentación con ChatGPT

Código Original

```
1 def proc_data(d):
2     r = []
3     for i in range(len(d)):
4         if isinstance(d[i], int):
5             if d[i] % 2 == 0:
6                 r.append(d[i] * 2)
7             else:
8                 r.append(d[i] + 1)
9         elif isinstance(d[i], str) and d[i].isdigit():
10            r.append(int(d[i]) + 100)
11        else:
12            continue
13    return sorted(r, reverse=True)
14
```

Caso Práctico 2: Agilizar la documentación con ChatGPT

Respuesta de ChatGPT

Código Original

```
1 def proc_data(d):
2     r = []
3     for i in range(len(d)):
4         if isinstance(d[i], int):
5             if d[i] % 2 == 0:
6                 r.append(d[i] * 2)
7             else:
8                 r.append(d[i] + 1)
9         elif isinstance(d[i], str) and d[i].isdigit():
10            r.append(int(d[i]) + 100)
11        else:
12            continue
13    return sorted(r, reverse=True)
14
```

Aquí tienes un docstring en estilo NumPy para la función proc_data:

```
def proc_data(d):
    """
    Procesa una lista de elementos transformando enteros y cadenas numéricas, y retorna un
    lista de enteros ordenados en orden descendente.

    Parámetros
    -----------
    d : list
        Lista de elementos que puedes incluir enteros, cadenas y otros tipos de datos.

    Devuelve
    -----------
    list
        Lista de enteros procesados y ordenadas en orden descendiente.

    Ejemplos
    ---------
    >>> proc_data([2, 3, '5', 'abc', 4])
    [10, 100, 4, 4]

    >>> proc_data(['10', 3, 'a', 'Hello', 0])
    [100, 0, 0, 0]

    >>> proc_data(['a', 0, -3])
    []
    """

    r = []
    for i in range(len(d)):
        if isinstance(d[i], int):
            if d[i] % 2 == 0:
                r.append(d[i] * 2)
            else:
                r.append(d[i] + 1)
        elif isinstance(d[i], str) and d[i].isdigit():
            r.append(int(d[i]) + 100)
        else:
            continue
    return sorted(r, reverse=True)
```

Pregunta lo quequieras

+ ➔ Herramientas

ChatGPT puede cometer errores. Considera verificar la información importante. Ver [https://openai.com](#)

Caso Práctico 3: Agilizar la documentación con LM Studio

Código Original

```
1 def proc_data(d):
2     r = []
3     for i in range(len(d)):
4         if isinstance(d[i], int):
5             if d[i] % 2 == 0:
6                 r.append(d[i] * 2)
7             else:
8                 r.append(d[i] + 1)
9         elif isinstance(d[i], str) and d[i].isdigit():
10            r.append(int(d[i]) + 100)
11        else:
12            continue
13    return sorted(r, reverse=True)
14
```

Caso Práctico 3: Agilizar la documentación con LM Studio

Respuesta de LM Studio

Código Original

```
1 def proc_data(d):
2     r = []
3     for i in range(len(d)):
4         if isinstance(d[i], int):
5             if d[i] % 2 == 0:
6                 r.append(d[i] * 2)
7             else:
8                 r.append(d[i] + 1)
9         elif isinstance(d[i], str) and d[i].isdigit():
10            r.append(int(d[i]) + 100)
11        else:
12            continue
13    return sorted(r, reverse=True)
14
```

```
def proc_data(d):
    """Procesa una lista de datos, aplicando transformaciones basadas en el tipo
    valor.
```

Esta función itera sobre una lista de datos ('d'). Para cada elemento, verifica si es un entero o una cadena que representa un número. Si es un entero, lo duplica si es par y le suma 1 si es impar. Si es una cadena que contiene solo dígitos, la convierte a entero y le suma 100. Los elementos que no cumplen con estas condiciones se ignoran. La función devuelve una nueva lista que contiene los resultados transformados, ordenada de forma descendente.

Parámetros

d : list
Una lista de datos que puede contener enteros y cadenas.

Retorno

list
Una nueva lista que contiene los resultados transformados, ordenada de forma descendente.

Ejemplos

```
>>> proc_data([1, 2, "3", 4, "abc"])
[100, 102, 6, 5]
```

```
>>> proc_data([1, 3, 5])
[6, 4, 2]
```

```
>>> proc_data([2, 4, 6])
[12, 8, 4]
```

```
>>> proc_data(["1", "2", "3"])
[100, 102, 301]
```

```
>>> proc_data(["2", "3", "4"])
[100, 103, 5, 23]
```

```
>>> proc_data([1, "a", 3])
[4, 2]
```

```
r = []
for i in range(len(d)):
    if isinstance(d[i], int):
        if d[i] % 2 == 0:
            r.append(d[i] * 2)
        else:
            r.append(d[i] + 1)
    elif isinstance(d[i], str) and d[i].isdigit():
        r.append(int(d[i]) + 100)
    else:
        continue
return sorted(r, reverse=True)
```

File: /tmp/llm-11344/proc_data.py | Line: 1 | Last update: 2023-07-10 10:45:40

Type a message and press Enter to send...:



User



Insert



Contact is 100% full

Limitaciones y validación

- LLMs pueden inventar explicaciones incorrectas.
- Es recomendable validar la documentación generada con pruebas.
- Puede ser útil usar prompts iterativos para mejorar claridad y precisión.

Buenas prácticas generales

- **Valida siempre el output del LLM** con tests u otras fuentes.
- **Reformula los prompts** si la respuesta no es útil o precisa.
- **Divide tareas complejas** en subtareas más simples con prompts encadenados.
- **Conserva el contexto relevante** del código y documentación al interactuar.

Consejos específicos por tarea

- Para refactorización: especifica qué se debe preservar (nombres, funcionalidad, rendimiento).
- Para tests: incluye ejemplos típicos y casos borde en el prompt.
- Para documentación: pide formato específico (Google, NumPy, etc.) y ejemplos.
- Aprovecha herramientas como Copilot o ChatGPT en modo interactivo para revisión incremental.

Resumen de Tareas Asistidas por LLMs



Reflexión y discusión final

¿Qué hemos aprendido hoy?

- Cómo los LLMs pueden asistir eficazmente en la refactorización de código.
- El uso de LLMs para generar pruebas unitarias de forma rápida y automatizada.
- La utilidad de los modelos para documentar código en distintos estilos.
- Buenas prácticas y riesgos a tener en cuenta al delegar tareas en un modelo.

Para debatir:

- ¿En qué tareas sientes que los LLMs te han aportado más valor?
- ¿En qué puntos aún confías más en tu propio criterio o en un humano?
- ¿Qué tareas automatizarías mañana mismo con un LLM?