

# Proyecto Python Refactorizado

## Estructura del Proyecto Mejorada

```
project/
├── src/
│   ├── __init__.py
│   ├── main.py
│   ├── math_operations.py
│   └── config.py
└── tests/
    ├── __init__.py
    └── test_math_operations.py
├── .gitignore
└── requirements.txt
└── requirements-dev.txt
└── README.md
```

### src/main.py

```
python
```

"""

Módulo principal del proyecto.

Este módulo procesa una lista de números y calcula sus cuadrados utilizando operaciones matemáticas del módulo math\_operations.

"""

```
import logging
import sys
from typing import List

from config import DEFAULT_DATA, LOG_FORMAT, LOG_LEVEL
from math_operations import square_number

# Configuración de logging
logging.basicConfig(
    level=LOG_LEVEL,
    format=LOG_FORMAT,
    handlers=[
        logging.StreamHandler(sys.stdout),
        logging.FileHandler('app.log')
    ]
)

logger = logging.getLogger(__name__)
```

```
def process_numbers(data: List[int | float]) -> List[int | float]:
```

"""

Procesa una lista de números y calcula sus cuadrados.

Args:

data: Lista de números (int o float) a procesar.

Returns:

Lista con los cuadrados de los números de entrada.

Raises:

ValueError: Si la lista está vacía o contiene valores no numéricos.

TypeError: Si el argumento no es una lista.

Examples:

```
>>> process_numbers([1, 2, 3])
```

```
[1, 4, 9]
>>> process_numbers([2.5, 3.5])
[6.25, 12.25]
"""

if not isinstance(data, list):
    logger.error(f"Se esperaba una lista, se recibió: {type(data)}")
    raise TypeError("El argumento debe ser una lista")

if not data:
    logger.warning("La lista de datos está vacía")
    raise ValueError("La lista no puede estar vacía")

# Validar que todos los elementos sean numéricos
for i, value in enumerate(data):
    if not isinstance(value, (int, float)):
        logger.error(f"Valor no numérico en índice {i}: {value}")
        raise ValueError(
            f"Todos los elementos deben ser números."
            f"Valor inválido en índice {i}: {value}"
        )

logger.info(f"Procesando {len(data)} números")

# List comprehension: más eficiente y pythonic
result = [square_number(x) for x in data]

logger.info(f"Procesamiento completado exitosamente")
return result
```

```
def main() -> None:
"""

Función principal del programa.
```

Ejecuta el procesamiento de números y muestra los resultados.

Maneja errores de forma robusta con logging apropiado.

"""

try:

```
    logger.info("Iniciando aplicación")
```

*# Usar datos de configuración*

```
    data = DEFAULT_DATA
```

```
    logger.debug(f"Datos de entrada: {data}")
```

```
# Procesar números
result = process_numbers(data)

# Mostrar resultados
print(f"Números originales: {data}")
print(f"Cuadrados calculados: {result}")

logger.info("Aplicación finalizada exitosamente")

except ValueError as e:
    logger.error(f"Error de validación: {e}")
    print(f"❌ Error: {e}", file=sys.stderr)
    sys.exit(1)

except TypeError as e:
    logger.error(f"Error de tipo: {e}")
    print(f"❌ Error: {e}", file=sys.stderr)
    sys.exit(1)

except Exception as e:
    logger.critical(f"Error inesperado: {e}", exc_info=True)
    print(f"❌ Error crítico: {e}", file=sys.stderr)
    sys.exit(1)

if __name__ == "__main__":
    main()
```

---

## src/math\_operations.py

```
python
```

!!!!

## Módulo de operaciones matemáticas.

Proporciona funciones utilitarias para operaciones matemáticas comunes con validación de tipos y manejo de errores.

!!!!

```
from typing import Union
```

# Type alias para mejorar legibilidad

```
Numeric = Union[int, float]
```

```
def square_number(x: Numeric) -> Numeric:
```

!!!!

Calcula el cuadrado de un número.

Args:

x: Número a elevar al cuadrado (int o float).

Returns:

El cuadrado del número de entrada. Retorna int si la entrada es int, float si la entrada es float.

Raises:

TypeError: Si el argumento no es un número (int o float).

ValueError: Si el resultado causa overflow.

Examples:

```
>>> square_number(5)
```

25

```
>>> square_number(2.5)
```

6.25

```
>>> square_number(-3)
```

9

!!!!

# Validación de tipo

```
if not isinstance(x, (int, float)):
```

```
    raise TypeError(
```

f"El argumento debe ser int o float, se recibió {type(x).\_\_name\_\_}"

```
)
```

# Validación de valores especiales

```
if x != x: # Detectar NaN
    raise ValueError("No se puede calcular el cuadrado de NaN")

try:
    # Usar operador ** que es más explícito que *
    result = x ** 2

    # Verificar overflow en floats
    if isinstance(result, float) and (result == float('inf') or result == float('-inf')):
        raise ValueError(f"Overflow al calcular el cuadrado de {x}")

    return result

except OverflowError:
    raise ValueError(f"El número {x} es demasiado grande para calcular su cuadrado")
```

**def cube\_number(x: Numeric) -> Numeric:**

!!!!

Calcula el cubo de un número.

Args:

x: Número a elevar al cubo (int o float).

Returns:

El cubo del número de entrada.

Raises:

TypeError: Si el argumento no es un número.

Examples:

>>> cube\_number(3)

27

>>> cube\_number(2.0)

8.0

!!!!

if not isinstance(x, (int, float)):

raise TypeError

f'El argumento debe ser int o float, se recibió {type(x).\_\_name\_\_}'

)

return x \*\* 3

```
def power_number(x: Numeric, exponent: Numeric) -> Numeric:
```

```
"""
```

Eleva un número a una potencia específica.

Args:

x: Número base (int o float).

exponent: Exponente (int o float).

Returns:

x elevado a la potencia del exponente.

Raises:

TypeError: Si los argumentos no son números.

ValueError: Si la operación resulta en un valor inválido.

Examples:

```
>>> power_number(2, 3)
```

```
8
```

```
>>> power_number(4, 0.5)
```

```
2.0
```

```
"""
```

```
if not isinstance(x, (int, float)) or not isinstance(exponent, (int, float)):
```

```
    raise TypeError("Ambos argumentos deben ser números")
```

```
try:
```

```
    return x ** exponent
```

```
except (ValueError, OverflowError) as e:
```

```
    raise ValueError(f"Error al calcular {x}^{exponent}: {str(e)}")
```

## src/config.py

```
python
```

.....

Configuración de la aplicación.

Centraliza constantes y parámetros configurables del proyecto.

.....

```
import logging
from typing import List

# Versión del proyecto
__version__ = "1.0.0"

# Datos por defecto para procesamiento
DEFAULT_DATA: List[int] = [1, 2, 3, 4, 5]

# Configuración de logging
LOG_LEVEL = logging.INFO
LOG_FORMAT = "%(asctime)s - %(name)s - %(levelname)s - %(message)s"

# Constantes matemáticas
MAX_INPUT_SIZE = 1000 # Máximo número de elementos a procesar
MIN_NUMBER_VALUE = -1e10 # Valor mínimo permitido
MAX_NUMBER_VALUE = 1e10 # Valor máximo permitido
```

---

## src/init.py

```
python
```

```
"""
```

```
Paquete principal del proyecto.
```

```
Expone las funciones principales para uso externo.
```

```
"""
```

```
from .config import __version__
from .math_operations import square_number, cube_number, power_number
from .main import process_numbers
```

```
__all__ = [
    "__version__",
    "square_number",
    "cube_number",
    "power_number",
    "process_numbers"
]
```

## tests/init.py

```
python
```

```
"""Paquete de tests unitarios."""
```

## tests/test\_math\_operations.py

```
python
```

```
"""
```

```
Tests unitarios para el módulo math_operations.
```

```
Verifica el correcto funcionamiento de las operaciones matemáticas.
```

```
"""
```

```
import pytest
from src.math_operations import square_number, cube_number, power_number


class TestSquareNumber:
    """Tests para la función square_number."""

    def test_positive_integer(self):
        """Verifica que funciona con enteros positivos."""
        assert square_number(5) == 25
        assert square_number(10) == 100

    def test_negative_integer(self):
        """Verifica que funciona con enteros negativos."""
        assert square_number(-5) == 25
        assert square_number(-10) == 100

    def test_zero(self):
        """Verifica que funciona con cero."""
        assert square_number(0) == 0

    def test_float(self):
        """Verifica que funciona con números decimales."""
        assert square_number(2.5) == 6.25
        assert square_number(0.5) == 0.25

    def test_invalid_type_string(self):
        """Verifica que rechaza strings."""
        with pytest.raises(TypeError):
            square_number("5")

    def test_invalid_type_list(self):
        """Verifica que rechaza listas."""
        with pytest.raises(TypeError):
            square_number([5])

    def test_invalid_type_none(self):
```

```
"""Verifica que rechaza None."""
with pytest.raises(TypeError):
    square_number(None)

class TestCubeNumber:
    """Tests para la función cube_number."""

    def test_positive_integer(self):
        """Verifica que funciona con enteros positivos."""
        assert cube_number(3) == 27
        assert cube_number(2) == 8

    def test_negative_integer(self):
        """Verifica que funciona con enteros negativos."""
        assert cube_number(-3) == -27

    def test_float(self):
        """Verifica que funciona con decimales."""
        assert cube_number(2.0) == 8.0

class TestPowerNumber:
    """Tests para la función power_number."""

    def test_basic_power(self):
        """Verifica potencias básicas."""
        assert power_number(2, 3) == 8
        assert power_number(5, 2) == 25

    def test_fractional_exponent(self):
        """Verifica exponentes fraccionarios."""
        assert power_number(4, 0.5) == 2.0
        assert power_number(27, 1/3) == pytest.approx(3.0)

    def test_zero_exponent(self):
        """Verifica exponente cero."""
        assert power_number(5, 0) == 1
        assert power_number(100, 0) == 1
```

## .gitignore

gitignore

```
# Python
__pycache__/
*.py[cod]
*$py.class
*.so
.Python
build/
develop-eggs/
dist/
downloads/
eggs/
.eggs/
lib/
lib64/
parts/
sdist/
var/
wheels/
*.egg-info/
.installed.cfg
*.egg
```

```
# Virtual Environment
venv/
env/
ENV/
```

```
# IDEs
.vscode/
.idea/
*.swp
*.swo
*~
```

```
# Testing
.pytest_cache/
.coverage
htmlcov/
```

```
# Logs
*.log
logs/
```

```
# OS  
.DS_Store  
Thumbs.db
```

---

## requirements.txt

```
txt  
  
# Dependencias de producción (actualmente ninguna necesaria)  
# Agregar aquí según necesidades futuras
```

---

## requirements-dev.txt

```
txt  
  
# Dependencias de desarrollo  
pytest>=7.4.0  
pytest-cov>=4.1.0  
pylint>=3.0.0  
black>=23.7.0  
mypy>=1.5.0  
flake8>=6.1.0
```

---

## README.md

```
markdown
```

## # 🧩 Mini Proyecto Python - Actividad 2.13

Proyecto Python refactorizado con mejores prácticas, validación de tipos, manejo de errores y tests unitarios.

### ## 🌟 Características

- Type hints completos
- Docstrings profesionales (Google style)
- Manejo robusto de errores
- Logging configurado
- Tests unitarios con pytest
- Código optimizado (list comprehensions)
- Validación de datos
- Estructura modular

### ## 📦 Instalación

```
```bash
# Clonar repositorio
git clone
cd project

# Crear entorno virtual
python -m venv venv
source venv/bin/activate # En Windows: venv\Scripts\activate
```

### # Instalar dependencias de desarrollo

```
pip install -r requirements-dev.txt
```

```
```
```

### ## 💡 Uso

```
```bash
# Ejecutar programa principal
python src/main.py
````
```

### ### Uso como módulo

```
```python
from src import square_number, process_numbers

# Calcular cuadrado de un número
result = square_number(5) # 25
```

```
# Procesar lista de números
numbers = [1, 2, 3, 4, 5]
squares = process_numbers(numbers) #[1, 4, 9, 16, 25]
```

```

### ## 🧪 Testing

```
```bash
# Ejecutar todos los tests
pytest tests/

```

### # Con cobertura

```
pytest --cov=src tests/
```

### # Con reporte HTML

```
pytest --cov=src --cov-report=html tests/
```

```

### ## 🔎 Linting y Formateo

```
```bash
# Formatear código con black
black src/ tests/

```

### # Análisis estático con pylint

```
pylint src/
```

### # Type checking con mypy

```
mypy src/
```

### # Style checking con flake8

```
flake8 src/
```

```

### ## 📁 Estructura del Proyecto

```
project/ └── src/ # Código fuente |   ├── init.py |   └── main.py # Punto de entrada |   └──
math_operations.py # Operaciones matemáticas |       └── config.py # Configuración |   └── tests/ # Tests unitarios
|   ├── init.py |       └── test_math_operations.py |   └── .gitignore |   └── requirements.txt # Dependencias
producción |   └── requirements-dev.txt # Dependencias desarrollo └── README.md
```

## ## 🎯 Mejoras Implementadas

### ### Calidad de Código

- List comprehensions en lugar de bucles manuales
- Type hints en todas las funciones
- Docstrings completos (Google style)
- Validación de tipos y datos

### ### Estructura

- Separación en módulos lógicos
- Archivo de configuración centralizado
- Tests unitarios con pytest
- Archivos de dependencias

### ### Estilo

- Logging en lugar de prints
- Nombres descriptivos de archivos/funciones
- Constantes en UPPER\_CASE
- Espaciado PEP 8

### ### Robustez

- Manejo completo de excepciones
- Validación de entrada
- Logging estructurado
- Exit codes apropiados

## ## 📄 Ejemplos

### ### Ejemplo 1: Uso básico

```
'''python
from src import square_number

result = square_number(10)
print(result) # 100
'''
```

### ### Ejemplo 2: Procesamiento de lista

```
'''python
from src import process_numbers

data = [2, 4, 6, 8]
results = process_numbers(data)
```

```
print(results) # [4, 16, 36, 64]
```

```
'''
```

```
### Ejemplo 3: Manejo de errores
```

```
'''python
```

```
from src import square_number
```

```
try:
```

```
    result = square_number("texto")
```

```
except TypeError as e:
```

```
    print(f'Error: {e}')
```

```
'''
```

## 🏗 Desarrollo

```
### Agregar nueva funcionalidad
```

1. Implementar función en módulo apropiado
2. Agregar type hints y docstring
3. Crear tests unitarios
4. Verificar con linter y type checker
5. Actualizar documentación

```
### Ejecutar validaciones completas
```

```
'''bash
```

```
# Script de validación completa
```

```
black src/ tests/ && \
```

```
pylint src/ && \
```

```
mypy src/ && \
```

```
pytest --cov=src tests/
```

```
'''
```

## 📄 Licencia

MIT License - ver archivo LICENSE para detalles

## 🚀 Autor

Tu Nombre - Actividad 2.13 Claude Code

---

⭐ Si te resultó útil, dale una estrella al repositorio

---

# Resumen de Mejoras Aplicadas

## Código Refactorizado

1. **Type hints** completos en todas las funciones
2. **Docstrings** profesionales estilo Google
3. **List comprehension** optimizada
4. **Validación robusta** de tipos y datos
5. **Manejo de errores** con try-except específicos

## Estructura Mejorada

6. **Módulo config.py** para configuración centralizada
7. **Tests unitarios** con pytest
8. **Organización en carpetas** src/ y tests/
9. **init.py** para convertir en paquete
10. **Requirements** separados (dev y prod)

## Buenas Prácticas

11. **Logging profesional** en lugar de prints
12. **Nombres descriptivos** (math\_operations.py vs utils.py)
13. **Constantes en mayúsculas** (DEFAULT\_DATA)
14. **.gitignore** completo
15. **README.md** exhaustivo con ejemplos

## Calidad Profesional

16. **Type alias** para mejor legibilidad (Numeric)
17. **Validaciones de edge cases** (NaN, overflow)
18. **Exit codes** apropiados (sys.exit)
19. **Logging a archivo** además de consola
20. **Funciones adicionales** (cube, power) para extensibilidad