Sesión 5: Depuración y análisis de código

Duración: 45 minutos

🎯 Objetivos de aprendizaje

Al finalizar esta sesión, el participante será capaz de:

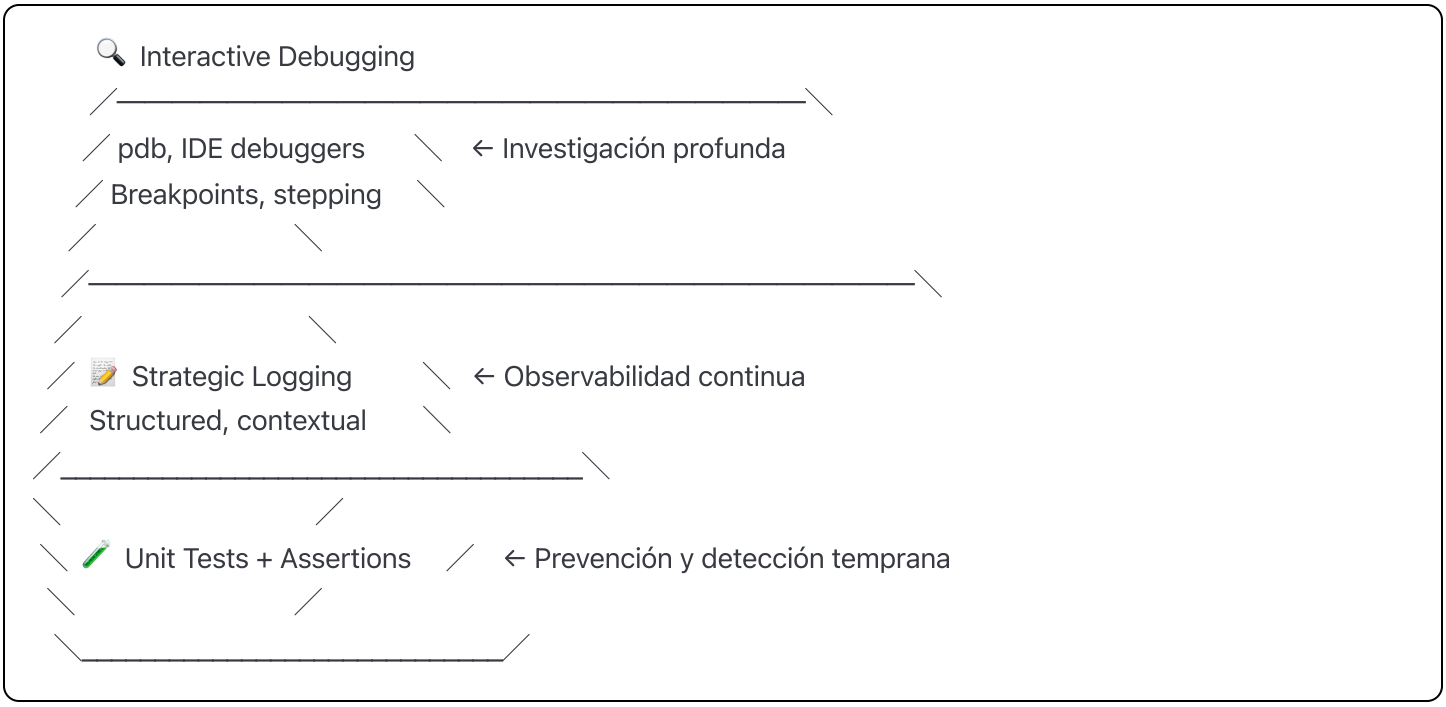
- Usar herramientas avanzadas de debugging en Python
- Implementar logging estratégico para debugging
- Realizar análisis estático de código con herramientas profesionales
- Identificar y resolver memory leaks y problemas de rendimiento
- Configurar profiling para optimización de código
- Integrar análisis de código en workflows de desarrollo

📋 Contenido

1. Estrategias de Debugging Moderno (8 min)

Analogía: El debugging es como ser detective. No basta con saber que algo está mal; necesitas evidencia, pistas, y un método sistemático para encontrar al culpable.

La pirámide del debugging:



Niveles de debugging:

1. **Preventivo:** Tests unitarios, assertions, type hints
2. **Observacional:** Logging estructurado, métricas
3. **Interactivo:** Debuggers, profilers, inspección en vivo
4. **Post-mortem:** Análisis de crashes, core dumps, logs

Mindset del debugging efectivo:

```
🎯 PROCESO SISTEMÁTICO:
├── 📋 Reproducir ──── Crear caso mínimo que reproduce el bug
├── 🔍 Aislar ──────── Reducir variables, usar debugging científico
├── 📊 Hipótesis ──── Formular teorías sobre la causa
├── 🧪 Validar ─────── Probar hipótesis sistemáticamente
└── 🔧 Arreglar ────── Implementar solución y verificar
```

2. Logging Estratégico para Debugging (10 min)

Levels de logging y su propósito:

```
🚨 CRITICAL ── Sistema no puede continuar
🔥 ERROR ──── Errores que deben ser arreglados
⚠️ WARNING ── Situaciones anómalas pero manejables
ℹ️ INFO ────── Flujo general del programa
🐛 DEBUG ──── Información detallada para debugging
```

Logging estructurado vs. no estructurado:

python

```python
# ❌ Logging no estructurado (difícil de analizar)
logger.info(f"User {username} logged in from {ip_address}")

# ✅ Logging estructurado (fácil de consultar/filtrar)
logger.info("User login successful", extra={
    'event': 'user_login',
    'username': username,
    'ip_address': ip_address,
    'timestamp': datetime.utcnow().isoformat(),
    'session_id': session_id
})
```

Contexto en logging:

```python
# Context managers para agregar contexto automático
@contextmanager
def log_context(**kwargs):
    # Agregar contexto al logger actual
    old_extra = getattr(logger, '_extra', {})
    logger._extra = {**old_extra, **kwargs}
    try:
        yield
    finally:
        logger._extra = old_extra

# Uso
with log_context(user_id=123, request_id='abc-def'):
    process_user_request()  # Todos los logs incluirán contexto
```

Estrategias de logging para debugging:

- **Entrada/salida de funciones** con parámetros y resultados
- **Estados de variables críticas** en puntos de decisión
- **Timing de operaciones** para detectar lentitud
- **Correlation IDs** para seguir requests a través del sistema

3. Herramientas de Debugging Interactivo (8 min)

pdb: Python Debugger nativo

```python
```

```python
import pdb

def problematic_function(data):
    result = []
    for item in data:
        pdb.set_trace()  # Breakpoint aquí
        processed = complex_processing(item)
        result.append(processed)
    return result

# Comandos básicos de pdb:
# l (list) - mostrar código actual
# n (next) - siguiente línea
# s (step) - entrar en funciones
# c (continue) - continuar ejecución
# p variable_name - mostrar valor de variable
# pp variable_name - pretty print
# w (where) - stack trace
# q (quit) - salir del debugger
```

pdb++ (pdbpp): Mejora del pdb tradicional

```bash
pip install pdbpp
# Automáticamente reemplaza pdb con versión mejorada
# + Syntax highlighting
# + Tab completion
# + Better stack traces
```

pudb: Debugger visual en terminal

```python
import pudb
pudb.set_trace()  # Interfaz visual completa

# Características:
# - Vista de código con highlighting
# - Variables panel
# - Stack trace visual
# - Breakpoints persistentes
```

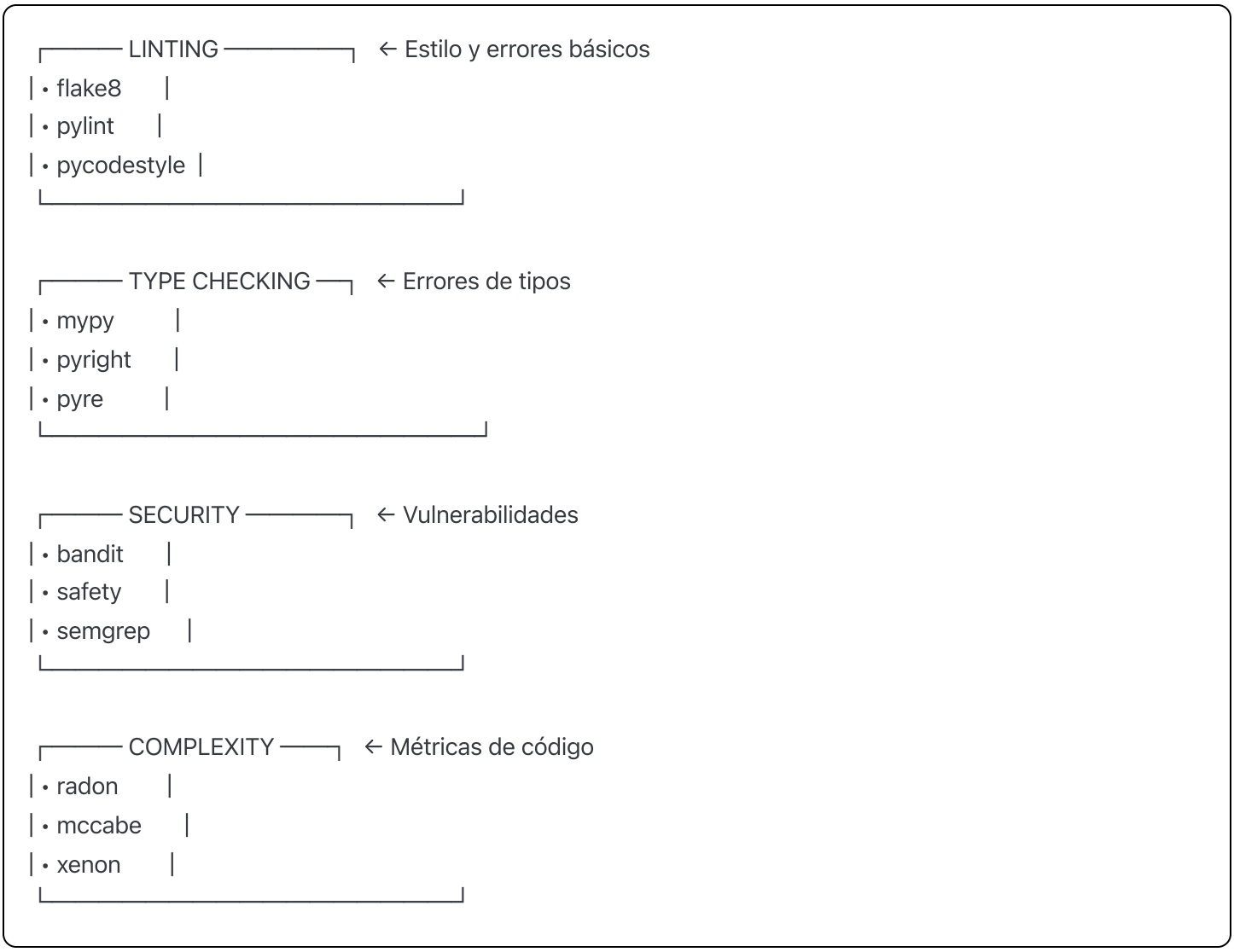
IDE Debugging:

- **VS Code:** Breakpoints, watch variables, call stack
- **PyCharm:** Advanced debugging, remote debugging
- **Debugger configurations** para diferentes scenarios

4. Análisis Estático de Código (10 min)

¿Qué es análisis estático? Análisis del código **sin ejecutarlo** para encontrar potenciales problemas, bad practices, y vulnerabilidades.

Herramientas principales:



Configuración integrada:

```toml
```

```
# pyproject.toml
[tool.flake8]
max-line-length = 88
extend-ignore = ["E203", "W503"]
exclude = [".git", "__pycache__", "migrations"]

[tool.mypy]
python_version = "3.9"
warn_return_any = true
warn_unused_configs = true
disallow_untyped_defs = true

[tool.bandit]
exclude_dirs = ["tests", "migrations"]
skips = ["B101"]  # Skip assert_used test
```

Pre-commit hooks para automatización:

yaml

```
# .pre-commit-config.yaml
repos:
  - repo: https://github.com/psf/black
    rev: 22.3.0
    hooks:
      - id: black
  - repo: https://github.com/pycqa/flake8
    rev: 4.0.1
    hooks:
      - id: flake8
  - repo: https://github.com/pre-commit/mirrors-mypy
    rev: v0.991
    hooks:
      - id: mypy
  - repo: https://github.com/PyCQA/bandit
    rev: 1.7.4
    hooks:
      - id: bandit
```

5. Profiling y Análisis de Rendimiento (7 min)

Tipos de profiling:

```
🕐 TIME PROFILING ──────── ¿Dónde se gasta el tiempo?
├─── cProfile ──────────────────── Built-in, statistical profiling
├─── py-spy ────────────────────── Production profiling, low overhead
└─── line_profiler ──────── Line-by-line timing

💾 MEMORY PROFILING ──── ¿Dónde se usa la memoria?
├─── memory_profiler ──── Line-by-line memory usage
├─── tracemalloc ──────────── Built-in memory tracking
└─── pympler ──────────────────── Advanced memory analysis

🔄 CALL PROFILING ──────── ¿Qué funciones se llaman más?
├─── cProfile ──────────────────── Call count and time
└─── pyinstrument ──────────── Statistical call profiling
```

cProfile básico:

python

```python
import cProfile
import pstats

# Profiling de función específica
pr = cProfile.Profile()
pr.enable()
your_function()
pr.disable()

# Análisis de resultados
stats = pstats.Stats(pr)
stats.sort_stats('cumulative')
stats.print_stats(10)  # Top 10 funciones
```

Memory profiling:

python

```python
from memory_profiler import profile

@profile
def memory_intensive_function():
    # Cada línea será medida
    data = [i for i in range(1000000)]  # +76.3 MiB
    processed = [x * 2 for x in data]   # +76.3 MiB
    del data                            # -76.3 MiB
    return processed
```

Continuous profiling en producción:

```python
# py-spy para profiling en vivo sin modificar código
# Terminal 1: Ejecutar aplicación
python your_app.py

# Terminal 2: Profiling en vivo
py-spy record -o profile.svg -- python your_app.py
py-spy top --pid <process_id>  # Live view
```

6. Debugging de Problemas Específicos (2 min)

Memory leaks:

```python
import tracemalloc

# Detectar memory leaks
tracemalloc.start()

# ... ejecutar código sospechoso ...

snapshot = tracemalloc.take_snapshot()
top_stats = snapshot.statistics('lineno')

for stat in top_stats[:10]:
    print(f"{stat.traceback.format()} - {stat.size / 1024 / 1024:.1f} MB")
```

Race conditions y concurrencia:

```python
# Logging thread-safe para debugging concurrente
import logging
import threading

logging.basicConfig(
    format='%(asctime)s [%(threadName)s] %(message)s',
    level=logging.DEBUG
)

def debug_with_thread_info(message):
    thread_id = threading.current_thread().ident
    logging.debug(f"[Thread {thread_id}] {message}")
```

🔧 Herramientas del Ecosistema

Debugging:

- `pdb` / `pdbpp` - Debugger interactivo
- `pudb` - Debugger visual
- `icecream` - Print debugging mejorado
- `snoop` - Tracing automático de ejecución

Static Analysis:

- `flake8` - Linting básico
- `pylint` - Análisis exhaustivo
- `mypy` - Type checking
- `bandit` - Security analysis
- `vulture` - Dead code detection

Profiling:

- `cProfile` - Time profiling built-in
- `py-spy` - Production profiling
- `memory_profiler` - Memory usage
- `pyinstrument` - Statistical profiler

Métricas y monitoreo:

- `psutil` - System resource monitoring
- `structlog` - Structured logging
- `sentry-sdk` - Error tracking
- `prometheus_client` - Metrics collection

⚠️ Mejores Prácticas

✅ DO:

- **Log context, not just events** → Include correlation IDs, user info
- **Use structured logging** → JSON format for easy querying
- **Profile before optimizing** → Measure, don't guess
- **Automate static analysis** → Pre-commit hooks, CI integration
- **Debug with scientific method** → Hypothesis → Test → Validate

❌ DON'T:

- **Leave debug prints in code** → Use proper logging levels
- **Profile in development only** → Monitor production performance
- **Ignore static analysis warnings** → They often catch real bugs
- **Debug without reproducing** → Always have a reliable repro case
- **Optimize without measuring** → Premature optimization is evil

---

📊 Workflow de Debugging Profesional

🐛 BUG REPORT
↓
📋 REPRODUCE LOCALLY
↓
🔍 GATHER EVIDENCE
├── Logs analysis
├── Stack traces
├── Environment info
└── User journey
↓
🧪 HYPOTHESIS FORMATION
↓
🔬 SYSTEMATIC TESTING
├── Unit tests for edge cases
├── Interactive debugging
├── Profiling if performance-related
└── Static analysis for code quality
↓
🔧 IMPLEMENT FIX
↓
✅ VALIDATE FIX
├── Automated tests
├── Manual testing
└── Performance regression check
↓
📦 DEPLOY & MONITOR

📝 Resumen Final

1. **Debugging sistemático** es más efectivo que debugging random
2. **Logging estructurado** proporciona observabilidad continua
3. **Análisis estático** detecta problemas antes de que lleguen a producción
4. **Profiling** identifica bottlenecks reales, no asumidos
5. **Herramientas integradas** en workflow previenen muchos bugs
6. **Automatización** de análisis asegura calidad consistente
7. **Monitoreo continuo** en producción detecta problemas temprano

---

🏃 Actividad Práctica Sugerida

Ejercicio 1 - Debugging:

1. Tomar código con un bug intencional
2. Usar pdb para investigar paso a paso
3. Implementar logging estructurado para observabilidad
4. Crear test que reproduce y valida la fix

Ejercicio 2 - Análisis:

1. Configurar pre-commit hooks con múltiples herramientas
2. Ejecutar análisis estático en código existente
3. Realizar profiling y identificar bottlenecks
4. Implementar optimizaciones basadas en datos

Tiempo estimado: 25-30 minutos adicionales