Debugging and Profiling

A Basic CS Skill, ABC Winter School

박원

Objective

- pdb, gdb
- fuzzing
- 프로파일링
  - cProfile, Valgrind

Debugging

- 프로그래밍을 완료하더라도 원치않는 동작이 발생할 수 있다.
- 이러한 원치 않는 동작을 버그라고 한다.
- 버그를 찾아내고, 고치는 일을 디버깅(Debugging)이라고 한다.

Debugging - Printing

- 코드 중간중간에 출력을 넣기
  - 가장 간단하고 가장 많이 사용하는 디버깅 방법
- 어디까지 프로그램이 도달하는지
  - 그때 변수 등의 상태는 어떤지
  - 를 알 수 있다.

Debugging - Printing

- 그러나 편하고 쉬운만큼 단점도 존재한다.
  - 매번 다시 컴파일을 해야할 필요가 있으며
  - 로우레벨에서는 프로그램의 동작이 달라질 수 있다.
  - 너무 많은 내용을 출력하게 될 수 있다.
- 그래서 디버거를 이용한다.

Debuger

- 디버거은 아래와 같은 디버깅을 위한 도구이다.
  - 특정 시점에 중단 시키는 일
  - 한줄씩 실행해서 상태를 보거나, 특정 위치로 jump
  - 변수 등의 상태를 확인
- C/C++ 같은 언어를 디버깅하는데에는 GDB를 쓴다
  - 이는 시스템프로그래밍 시간에 사용법을 배운다.
- 이번 시간에는 pdb를 이용해서 python을 디버깅한다.

PDB

- 우선 아래 프로그램을 작성해봅시다.

```
# myscript.py
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(n):
            if arr[j] > arr[j+1]:
                arr[j] = arr[j+1]
                arr[j+1] = arr[j]
    return arr

print(bubble_sort([4, 2, 1, 8, 7, 6]))
```

PDB

- pdb로 디버깅하고 싶다면 아래와 같이 실행합니다.

```
python3 -m pdb myscript.py
```

PDB

- 아래 기능이 있습니다.
  - `l` : 코드를 표시한다
  - `s` : 멈출 수 있는 다음 줄 까지 실행한다.
  - `n` : 한줄 단위로 실행하며, 함수라면 호출후 결과 반환까지 기다린다.
  - `b` : 정지 지점을 만든다
  - `p` : 현재 내용을 출력한다
  - `r` : 함수의 반환을 기다린다
  - `q` : 디버거를 종료한다

PDB - Exercise

- 한번 pdb를 이용하여 `myscript.py` 의 어느 부분에 오류가 있는지 확인하고, 그 오류를 고쳐봅시다.

Fuzzing

- 퍼징은 자동화된 버그 찾는 도구이다.
- 버그가 발생할 때 까지 무작위로 입력을 넣습니다.
- 예상치 못한 입력에 버그가 발생할 수 있음을 알 수 있습니다.
  - 최근 fuzzing을 이용하는 사례는 점점 늘고 있습니다.

정적 분석

- 프로그램을 실행하지 않고 버그를 찾을 수 있습니다.
- 이 경우에 훨씬 시간과 비용이 적게 버그를 찾을 수 있습니다.
- 이를 정적 분석이라고 합니다.
- 관련 내용은 프로그래밍언어와 소프트웨어공학을 수강하면 배워볼 수 있습니다.

정적 분석

- 주로 `pyflake`와 `mypy`를 이용합니다.

정적 분석

- 아래 코드를 분석해봅시다.

```
import time

def foo():
    return 42

for foo in range(5):
    print(foo)
bar = 1
bar *= 0.2
time.sleep(60)
print(baz)
```

Profiling

- 프로파일링은 프로그램의 버그를 찾고 수정하는 디버깅과 다릅니다.
- 프로파일링은 프로그램이 얼마나 오래 걸리고, 메모리는 얼마나 사용하며, 프로그램의 각 부분이 cpu를 얼마나 점유하고 있는지 확인할 수 있습니다.
- 즉 프로그램의 잘 동작하는지가 아니라, 프로그램이 **어떻게** 동작하고 있는지 확인하는 것 입니다.

Profiling - Time

- 프로그램의 어떤 부분이 시간을 얼마나 차지하는지는 time stamp를 찍는 코드를 추가하여 측정가능합니다.

```
import time, random
n = random.randint(1, 10) * 100

start = time.time()

print("Sleeping for {} ms".format(n))
time.sleep(n/1000)

print(time.time() - start)
```

Profiling - Time

- UNIX 환경이면 `time`을 이용할 수도 있습니다.

```
time python3 myscript.py
```

Profiling - CPU

- 한단계 아래로 내려가서 코드의 각 부분이나 함수가 cpu를 얼마나 점유하고 있는지 확인하고 싶습니다.
- 그럴때는 `cProfile`을 이용합니다.

```python
# grep.py
import sys, re

def grep(pattern, file):
    with open(file, 'r') as f:
        print(file)
        for i, line in enumerate(f.readlines()):
            pattern = re.compile(pattern)
            match = pattern.search(line)
            if match is not None:
                print("{}: {}".format(i, line), end="")

if __name__ == '__main__':
    times = int(sys.argv[1])
    pattern = sys.argv[2]
    for i in range(times):
        for file in sys.argv[3:]:
            grep(pattern, file)
```

Profiling - CPU

- `cProfile`은 정적 분석과 똑같이 아래처럼 사용할 수 있습니다.

```
python -m cProfile -s tottime grep.py 1000 '^(import|\s*def)[^,]*$' *.py
```

Profiling - Memory

- 프로그램의 동작에 있어서 메모리 사용량은 프로그램의 동작 시간 만큼이나 중요합니다.
- 다행인건 Python은 언어에 자체적으로 메모리 관리를 해주는 기능이 있습니다.
  - 하지만 C/C++은 직접 메모리 할당과 해제를 해줘야합니다
  - 이를 `Valgrind`라는 것으로 확인합니다.
- 파이썬은 `memory_profiler`를 이용합니다.

```python
# myscript.py
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(n):
            if arr[j] > arr[j+1]:
                arr[j] = arr[j+1]
                arr[j+1] = arr[j]
    return arr

print(bubble_sort([4, 2, 1, 8, 7, 6]))
```

```
$ python -m memory_profiler example.py
```

Profiling - Memory

Questions?

Next