# Debug

## Debugging

### assert

- ??????????
- ??????????
- `assert <condition>, <message>` : ?? condition ? True????????? False ????????????
- ???? `assert` ??????? `python -O sample.py`?

??????

```
x = 5
assert x == 5, "x should be 5"


assert type(username) == str, "username must be a string"
```

```
def calculate_square_root(x):
    assert x >= 0, "The input must be non-negative."
    return x ** 0.5

print(calculate_square_root(4))  # □□: 2.0
print(calculate_square_root(-1)) # □□ AssertionError □□□ "The input must be non-negative."
```

???????????

```
def get_even_number(numbers):
    for num in numbers:
        if num % 2 == 0:
            return num
    assert False, "No even number found in the list."

numbers = [1, 3, 5, 7, 10]
print(get_even_number(numbers))  # □□: 10
numbers = [1, 3, 5, 7]
```

```
print(get_even_number(numbers))  # □□ AssertionError □□□ "No even number found in the list."
```

## prinf debugging

```
print("Processing {}".format(basename))
```

## strace

- Linux strace Command Tutorial for Beginners (8 Examples)

```
# Installation on RHEL if it's not installed
yum install strace

# Tracing system calls made by a program
strace ./my-program.py
strace -o my-program.strace ./my-program
```

# Crash

## pdb

???

- ???????
- ???????
- ????
- ??????????

```
pdb3 myprog.py
```

pdb-subcommands

- continue : ????????????
- print() : ???????

```
(Pdb) continue
...
(Pdb) print(row)
```

Step 1: Set a breakpoint

```
import pdb



def add_numbers(a, b):
    pdb.set_trace()  # This will set a breakpoint in the code
    result = a + b
    return result



print(add_numbers(3, 4))
```

Setp 2: Enter the interactive debugger

- *a* (args): Show the arguments of the current function.
- *b:* Manually set a persistent breakpoint while in debugger.
- *n* (next): Execute the next line within the current function.
- *s* (step): Execute the current line and stop at the first possible occasion (e.g., in a function that is called).
- *c* (continue): Resume normal execution until the next breakpoint.
- *p* (print): Evaluate and print the expression, e.g., p *variable_name* will print the value of *variable_name*.
- *Pp* (pretty-print): Pretty-print the value of the expression.
- *q* (quit): Exit the debugger and terminate the program.
- *r* (return): Continue execution until the current function returns.
- *tbreak:* Manually set a temporary breakpoint that goes away once hit the first time.
- *!*: Prefix to execute an arbitrary Python command in the current environment, e.g., *!variable_name* = "*new_value*" will set variable_name to "*new_value*".

Step 3: Inspect variables

To inspect the variables, simply type the single character, *p*, then the variable name to see its current value. For instance, if you have a variable in your code named *sentiment_score*, just type p *sentiment_score* at the *pdb* prompt to inspect its value.

Step 4: Modify variables

A big advantage of pdb is that you can change the value of a variable directly in the debugger. For example, to change `sentiment_score` to 0.9, you'd type `!sentiment_score = 0.9`.

To confirm these changes, use a or directly probe the value with `p <value name>`.

Step 5: Exit the debugger

When you’re done, simply enter `q` (quit) to exit the debugger and terminate the program.

Post-mortem debugging

```
python -m pdb your_script.py
```

# Memory Leaks

??????????????????????????????????????????????????

## memory_profiler

????????????????????????????????????

```
python3 -m memory_profiler myprog.py
```

In Code

- ? `main()` ???? `@profile` ??
- @ ???? Decorator: ? Python ??????????????????????
- memory-profiler

```
from memory_profiler import profile

...
...

@profile
def main():
  ...
  ...
```

Revision #11
Created 2024-12-20 14:27:55 CST by A-Lang (Admin)
Updated 2025-12-17 13:22:46 CST by A-Lang (Admin)