Lab 5
Debugging and Documentation

COMS 2132 - Professor Bauer
TA Session by Darien Moment and Julien Remy

Debugging

What is Debugging?

- The process of identifying and fixing errors in code.
- Helps ensure programs run correctly and efficiently.

Why is Debugging Important?

- Prevents crashes and unexpected behavior.
- Saves time by catching issues early.
- Improves code reliability and performance.

Good debugging makes you a better programmer!

Print Statements

The simplest and easiest way to debug a program is using print statements. This process involves strategically placing print statements in the program to detect where or why a problem might occur. We can use print statements to print out execution paths, variable values, or other relevant information.

```
def division(a, b):
    if b < 0:
        print(f"In division with arguments: {a}, {b}")

    # ... some code

    div = a / b

    if div >= 0:
            print(f"Exiting division with result: {div}")
    return div
```

Print Statements

When we are trying to debug a program with loops, it might also be helpful to print iteration details.

```
i = 10
while i >= 0:
    print(f"i = {i}")
    if i % 2 == 0:
        i = i // 2
    else:
        i -= 1
```

Using print statements can be tedious in large programs.

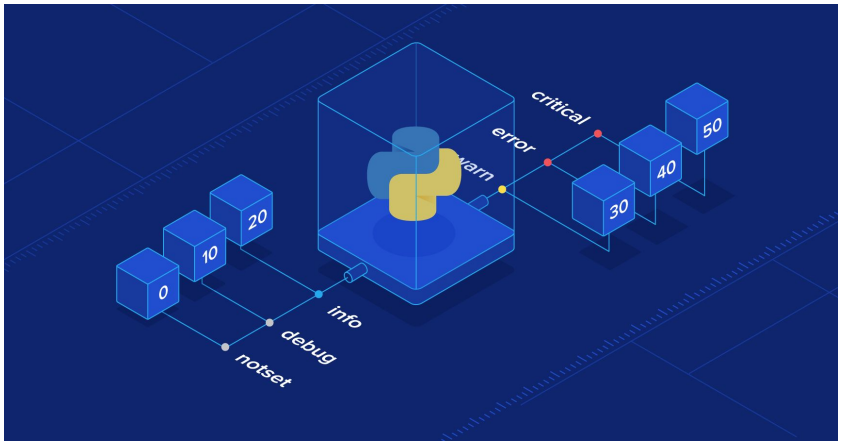
When adding debugging print statements to an existing program, be careful not to modify the program's behavior!

Logging Module

Python also offers the ‘logging’ module, which allows you to log messages with **different levels of severity or importance**

This is better than print statements because the debugging output can be selectively enabled and disabled **without the need to modify** the Python program

The two ways to selectively enable and disable logging messages are by **logging level** or **subsystem**



Logging by Levels

The logging module can log messages at the following logging levels (in increasing order of importance or severity): debug, info, warning, error, critical

```
import logging

logging.basicConfig(level=logging.DEBUG)

def check_negative(value):
    if value < 0:
        logging.warning("A negative value was passed!")
    else:
        logging.info("Operation performed successfully.")

some_value = 10

check_negative(some_value)
```

```
$ python main.py
INFO:root:Operation performed successfully.
```

The level argument passed to the function logging.basicConfig() determines the minimum level to enable.

- '**logging.DEBUG**' all messages will be logged
- '**logging.WARNING**' only warning, error, and critical messages will be logged

When you are debugging, you can change the minimum level to info or debug, depending on how much information you need.

Logging by Subsystem

The logging module can log messages from specific subsystems (Python Modules). You can define unique identifiers for subsystems, and selectively enable or disable the loggers.

// checker.py

```
import logging
# Create a new independent logger for the Python module
called 'checker'.
logger = logging.getLogger('checker')

def check_negative(value):
    logger.info(f"Running check_negative with {value}")
    if value < 0:
        logger.warning("Value cannot be negative.")
    else:
        logger.info("Operation performed successfully.")
```

// main.py

```
import logging
import checker

# Create a new logger for the main Python
module
logger = logging.getLogger('main')
logging.basicConfig(level=logging.WARNING)

some_value = -10

checker.check_negative(some_value)
logger.info('Call to check_negative
succeeded')
```

```
$ python main.py
WARNING:checker:Value cannot be negative.
```

Logging by Subsystem

// checker.py

```
import logging
# Create a new independent logger for the Python module
called 'checker'.
logger = logging.getLogger('checker')


def check_negative(value):
    logger.info(f"Running check_negative with {value}")
    if value < 0:
        logger.warning("Value cannot be negative.")
    else:
        logger.info("Operation performed successfully.")
```

// main.py

```
import logging
import checker

# Create a new logger for the main Python module
logger = logging.getLogger('main')
logging.basicConfig(level=logging.WARNING)

# Obtain a reference to the logger 'checker' and set
its logging level to DEBUG
logging.getLogger('checker').setLevel(logging.DEBUG)

some_value = -10

checker.check_negative(some_value)
logger.info('Call to check_negative succeeded')
```

```
$ python main.py
INFO:checker:Running check_negative with -10
WARNING:checker:Value cannot be negative
```

Exceptions

Exceptions are runtime errors that occur during execution. We can catch and handle exceptions during runtime, which can be useful in understanding the underlying issues with our program.

```
def divide(a, b):
    try:
        div = a / b
        return div
    except ZeroDivisionError as e:
        print(f"Error: Division by zero!")
        return None
    except Exception as e:
        print(f"Some other unexpected error occurred")
        return None
```

One of the ways to handle exceptions is by using try-except blocks. This way, we can catch exceptions and perform specified actions.

Custom Exceptions

We can also create custom exceptions in Python to catch application-specific errors.

```
class CustomError(Exception):
      def __init__(self, message):
          self.message = message
          super().__init__(message)

def check_for_negative(value):
       try:
           if value < 0:
               raise CustomError("Negative values are not allowed")
           return value
       except CustomError as ce:
           print(f"Custom error: {ce}")
           return abs(value)
       except Exception as e:
           print(f"Some other error occured.")
```

```
$ python main.py
Custom error: Negative values are not allowed
```

Traceback

The traceback module allows extraction, formatting, and printing of stack traces of Python programs.

Stack traces - the sequence of function calls that led to an exception or an error

```python
import traceback

def function_c():
    result = 10 / 0

def function_b():
    function_c()

def function_a():
    function_b()

try:
    function_a()
except Exception:
    traceback.print_exc(limit=2) #prints only the last two frames
```

```
$ python main.py
Traceback (most recent call last):
  File "main.py", line 13, in <module>
    function_a()
  File "main.py", line 10, in function_a
    function_b()
ZeroDivisionError: division by zero
```

The Python Debugger

Python comes with an interactive debugger module called “pdb”, which is useful for two things:

1. Stepping through the source code statement by statement
2. Setting a breakpoint in the program

Stepping through a program allows you to run it line by line, inspecting how variables change after each line is executed.

Setting a breakpoint allows you to enter the debugger once your program has reached a certain state or condition. The Python interpreter then stops running the program and enters a debugger session where you can tell it what to do next.

Stepping Through the Program

Suppose you are working on a program to print the Fibonacci series, you call the Fibonacci function with the value 10, and you want to “step through” the program to see how it behaves.

You can start the program through the “pdb” module as follows:

```
def fib(n):
    if n == 0 or n == 1:
        return n
    return fib(n - 1) + fib(n - 2)

if __name__ == '__main__':
  print(fib(10))
```

```
$ python -m pdb main.py
-> def fib(n):
(Pdb) n
> /Users/darienmoment/Desktop/main.py(6)<module>()
-> if __name__ == '__main__':
(Pdb) n
> /Users/darienmoment/Desktop/main.py(7)<module>()
-> print(fib(10))
(Pdb) n
55
--Return--
> /Users/darienmoment/Desktop/main.py(7)<module>()->None
-> print(fib(10))
(Pdb) n
--Return--
> <string>(1)<module>()->None
```

Stepping Through the Program

The “n(ext)” pdb command “steps over” the functions being used by the current statement.

Instead of stepping over the call to fib(10), we would have wanted to “step inside” the function so that we could debug it. There is another pdb command, “s(tep)” (for step in)

Let’s re-rerun the above session until we get to the print(fib(10)) statement, and then input “s(tep)” instead of “n(next)”:

```
$ python -m pdb main.py
-> def fib(n):
(Pdb) n
> /Users/darienmoment/Desktop/main.py(6)<module>()
-> if __name__ == '__main__':
(Pdb) n
> /Users/darienmoment/Desktop/main.py(7)<module>()
-> print(fib(10))
(Pdb) s
--Call--
```

```
> /Users/darienmoment/Desktop/main.py(1)fib()
-> def fib(n):
(Pdb) n
> /Users/darienmoment/Desktop/main.py(2)fib()
-> if n == 0 or n == 1:
(Pdb) n
> /Users/darienmoment/Desktop/main.py(4)fib()
-> return fib(n - 1) + fib(n - 2)
(Pdb) n
```

Setting a Breakpoint

Often, it is better to let the program run as normal and enter the debugger when a certain condition is met. **This is what breakpoints are for**

Suppose you only wanted to enter the debugger in the fib() function when it is passed the value 7 or 6. You can modify your program to check for the parameter value and add a breakpoint as follows (breakpoint is a reserved word in Python)

```
def fib(n):
    if n == 6:
        breakpoint()
    if n == 7:
        breakpoint()
    if n == 0 or n == 1:
        return n
    return fib(n - 1) + fib(n - 2)

if __name__ == '__main__':
    print(fib(10))
```

```
$ python -m pdb main.py
> /Users/darienmoment/Desktop/main.py(6)fib()
-> if n == 0 or n == 1:
(Pdb) p(n)
7
(Pdb) c
> /Users/darienmoment/Desktop/main.py(4)fib()
-> if n == 7:
(Pdb) p(n)
6
```

Documentation

What is Documentation?

- Written explanations that describe how code works.
- Includes comments, docstrings, and external guides.

Why Use It?

- Improves Readability – Easier to understand and use.
- Simplifies Debugging – Helps fix and update code.
- Supports Collaboration – Makes teamwork smoother.
- Essential for Large Projects – Ensures long-term usability.

Well-documented code saves time and effort!

Docstrings

Docstrings, short for documentation strings, are necessary to provide information about classes, modules or functions. Think of them as enhanced comments.

To create a docstring, simply use ‘’’triple single quotes’’’ or “””triple double quotes”””

```
def fib(n):
    """
    Calculate the nth Fibonacci number using recursion.

    Parameters:
    n (int): The position in the Fibonacci sequence (must be a non-negative integer).

    Returns:
    int: The nth Fibonacci number.
    """
    if n == 0 or n == 1:
        return n

    return fib(n - 1) + fib(n - 2)
```

Pydoc

PyDoc is a built-in module in Python that automatically generates documentation from docstrings. It can display documentation in the terminal or generate HTML pages.

```
$ pydoc main
NAME
    main

FUNCTIONS
    fib(n)
        Calculate the nth Fibonacci number using recursion.

        Parameters:
        n (int): The position in the Fibonacci sequence (must be a non-negative integer).

        Returns:
        int: The nth Fibonacci number.

FILE
    /Users/darienmoment/Desktop/main.py
```

Run **pydoc -w main** to generate an HTML file.