

DEBUGGING

CS 3080: Python Programming



University of Colorado
Colorado Springs

Debugging

- Now that you know enough to write more complicated programs, you may start finding not-so-simple bugs in them.
- This chapter covers some tools and techniques for finding the root cause of bugs in your program to help you fix bugs faster and with less effort.
- Definition (from google dictionary):
 - *The process of identifying and removing errors from computer hardware or software.*

Debugging

- Your computer will do only what you tell it to do; it won't read your mind and do what you intended it to do.
- Even professional programmers create bugs all the time, so don't feel discouraged if your program has a problem.



**CODE FOR SIX
MINUTES**

**DEBUG FOR SIX
HOURS**

memegenerator.net

Raising Exceptions

- Python raises an exception whenever it tries to execute invalid code
- But you can also **raise your own exceptions** in your code.
- Raising an exception is a way of saying:
 - *“Stop running the code in this function and move the program execution to the except statement.”*

Raising Exceptions

```
raise Exception('This is the error message.')  
# Traceback (most recent call last):  
# File "<pyshell#191>", line 1, in <module>  
# raise Exception('This is the error message.')  
# Exception: This is the error message.
```

Raising Exceptions

```
def boxPrint(symbol, width, height):  
    if len(symbol) != 1:  
        raise Exception('Symbol must be a single character string.')    if width <= 2:  
        raise Exception('Width must be greater than 2.')    if height <= 2:  
        raise Exception('Height must be greater than 2.')    print(symbol * width)  
  
    for i in range(height - 2):  
        print(symbol + (' ' * (width - 2)) + symbol)  
  
    print(symbol * width)
```

Raising Exceptions

```
for sym, w, h in (('*', 4, 4),
                  ('0', 20, 5),
                  ('x', 1, 3),
                  ('ZZ', 3, 3)):

    try:
        boxPrint(sym, w, h)
    except Exception as err:
        print('An exception happened: ' + str(err))
```


Getting the Traceback as a String

- When Python encounters an error, it produces a treasure trove of error information called the **traceback**.
- The traceback includes:
 - *the error message,*
 - *the line number of the line that caused the error,*
 - *and the sequence of the function calls that led to the error. This sequence of calls is called the **call stack**.*

Getting the Traceback as a String

```
def spam():  
    bacon()
```

```
def bacon():  
    raise Exception('This is the error message.')
```

```
spam()
```

Getting the Traceback as a String

```
def spam():  
    bacon()
```

```
def bacon():  
    raise Exception('This is the error message.')
```

```
spam()
```

```
Traceback (most recent call last):  
  File "/Users/damiafuentes/Documents/UCCS/CS 3030 Python/Lectures/test.py", line 7, in <module>  
    spam()  
  File "/Users/damiafuentes/Documents/UCCS/CS 3030 Python/Lectures/test.py", line 2, in spam  
    bacon()  
  File "/Users/damiafuentes/Documents/UCCS/CS 3030 Python/Lectures/test.py", line 5, in bacon  
    raise Exception('This is the error message.')  
Exception: This is the error message.
```

Getting the Traceback as a String

- Instead of crashing your program right when an exception occurs, you can write the traceback information to a log file and keep your program running. You can look at the log file later, when you're ready to debug your program.

```
import traceback
try:
    raise Exception('This is the error message.')
except:
    errorFile = open('errorInfo.txt', 'a')
    errorFile.write(traceback.format_exc())
    errorFile.close()
    print('The traceback info was written to errorInfo.txt.')
```

Assertions

- An assertion is a sanity check to make sure your code isn't doing something obviously wrong.
- These sanity checks are performed by **assert statements**.
- If the sanity check fails, then an `AssertionError` exception is raised.

Assertions

```
def kelvinToFahrenheit(temp):  
    assert (temp >= 0), "Colder than absolute zero!"  
    return ((temp-273)*1.8)+32
```

```
print(kelvinToFahrenheit(273))           # 32.0  
print(int(kelvinToFahrenheit(505.78)))   # 451  
print(kelvinToFahrenheit(-5))  
# AssertionError: Colder than absolute zero!
```

Assertions

- In plain English:
 - *“I assert that this condition holds true, and if not, there is a bug somewhere in the program.”*
- Unlike exceptions, your code should not handle assert statements with try and except; if an assert fails, your program should crash.
- By failing fast early in the program’s execution, you can save yourself a lot of future debugging effort.
- This will reduce the amount of code you will have to check before finding the code that’s causing the bug.
- **Assertions are for programmer errors, not user errors.** For errors that can be recovered (such as a file not being found or the user entering invalid data), raise an exception instead of detecting it with an assert statement.

Assertions

```
def kelvinToFahrenheit(temp):  
    assert (temp >= 0), "Colder than absolute zero!"  
    return ((temp-273)*1.8)+32
```

```
# Some code that obtains the kelvin temperature not  
# involving user input  
fahrenheit = kelvinToFahrenheit(kelvin)  
# Work with the fahrenheit temperature
```


Logging

- If you've ever put a `print()` statement in your code to output some variable's value while your program is running, you've used a form of logging to debug your code.
- Logging is a great way to understand what's happening in your program and in what order its happening.
- We are going to use the logging module

Logging

Where is the error?

```
import logging
```

```
logging.basicConfig(level=logging.DEBUG, format=' %(asctime)s - %(levelname)s  
- %(message)s')
```

```
logging.debug('Start of program')
```

```
def factorial(n):
```

```
    logging.debug('Start of factorial(%s%%)' % (n))
```

```
    total = 1
```

```
    for i in range(n + 1):
```

```
        total *= i
```

```
        logging.debug('i is ' + str(i) + ', total is ' + str(total))
```

```
    logging.debug('End of factorial({})'.format(n))
```

```
    return total
```

```
logging.debug(factorial(5))
```

```
logging.debug('End of program')
```

LogRecord attributes

```
logging.basicConfig(level=logging.DEBUG, format=' %(asctime)s  
- %(levelname)s - %(message)s')
```

```
2019-02-18 19:02:17,336 - DEBUG - Start of program  
2019-02-18 19:02:17,337 - DEBUG - Start of factorial(5%)  
2019-02-18 19:02:17,337 - DEBUG - i is 0, total is 0  
2019-02-18 19:02:17,337 - DEBUG - i is 1, total is 0  
2019-02-18 19:02:17,337 - DEBUG - i is 2, total is 0  
2019-02-18 19:02:17,337 - DEBUG - i is 3, total is 0  
2019-02-18 19:02:17,337 - DEBUG - i is 4, total is 0  
2019-02-18 19:02:17,337 - DEBUG - i is 5, total is 0  
2019-02-18 19:02:17,337 - DEBUG - End of factorial(5)  
2019-02-18 19:02:17,337 - DEBUG - 0  
2019-02-18 19:02:17,337 - DEBUG - End of program
```

LogRecord attributes

- `%(asctime)s`
 - *Human-readable time*
- `%(filename)s`
- `%(pathname)s`
- `%(funcName)s`
- `%(levelname)s`
 - *The logging level for the message ('DEBUG', 'INFO', 'WARNING', 'ERROR', 'CRITICAL').*
- `%(lineno)d`
 - *Source line number where the logging call was issued (if available)*
- `%(message)s`
 - *The logged message*

Find all of them: <https://docs.python.org/3/library/logging.html#logrecord-attributes>

Logging levels

Level	Logging Function	Description
DEBUG	<code>logging.debug()</code>	The lowest level. Used for small details. Usually you care about these messages only when diagnosing problems.
INFO	<code>logging.info()</code>	Used to record information on general events in your program or confirm that things are working at their point in the program.
WARNING	<code>logging.warning()</code>	Used to indicate a potential problem that doesn't prevent the program from working but might do so in the future.
ERROR	<code>logging.error()</code>	Used to record an error that caused the program to fail to do something.
CRITICAL	<code>logging.critical()</code>	The highest level. Used to indicate a fatal error that has caused or is about to cause the program to stop running entirely.

Logging levels

- After developing your program some more, you may be interested only in errors. In that case, you can set `basicConfig()`'s `level` argument to `logging.ERROR`. This will show only `ERROR` and `CRITICAL` messages and skip the `DEBUG`, `INFO`, and `WARNING` messages.

```
logging.basicConfig(level=logging.ERROR, format='%(asctime)s  
- %(levelname)s - %(filename)s - %(lineno)d - %(message)s')
```

Disabling Logging

- After you've debugged your program, you probably don't want all these log messages cluttering the screen.
- The `logging.disable()` function disables these so that you don't have to go into your program and remove all the logging calls by hand.
- You simply pass `logging.disable()` a logging level, and it will suppress all log messages at that level or lower.
- So if you want to disable logging entirely, just add
 - `logging.disable(logging.CRITICAL)`

Logging to a File

- While logging messages are helpful, they can clutter your screen and make it hard to read the program's output. Writing the logging messages to a file will keep your screen clear and store the messages so you can read them after running the program.

```
import logging
```

```
logging.basicConfig(filename='myProgramLog.txt',  
                    level=logging.DEBUG,  
                    format='%(asctime)s - %(levelname)s - %(message)s')
```