Chapter 9 - Debugging, Testing and Profiling

CS 172 - Computer Programming 2
Lanzhou University

These slides use many elements provided in the main bibliographic reference for these lectures:

Programming in Python 3
A Complete Introduction to the Python Language,
2nd Edition,
Mark Summerfield

Outline

Why are Debugging, Testing and Profiling needed?

- Humans make mistakes in their day-to-day tasks;
- Being done by humans, programming is particularly mistake-prone:
  - it is a *complex* task,
  - which involves art, craft, science and (lots of) *training*;
- Mistakes in a program:
  - may prevent it from being executed
    - it is not possible to understand which program was written;
  - may not terminate and/or lead to the production of wrong outputs
    - which can actually have tremendous impacts (and actually cost lives!);

Why are Debugging, Testing and Profiling needed?

- Programming mistakes fall in different categories:
  - syntax errors
    - prevent the program from compiling/being interpreted and running;
    - quickest to reveal, by the compiler;
    - easiest to fix;
    - usually due to typos (e.g., typo in the name of a class or variable);
  - logical errors
    - the program runs,
    - but some aspect of its behavior is not the intended one, that is, not according to the requirements;
  - performance issues
    - the program runs and terminates;
    - and actually behaves as intended,
    - but has needlessly poor performance;
    - this is usually due to a poor choice of algorithm and/or data structure;

Why are Debugging, Testing and Profiling needed?

About Logical Errors

- They are typically the most challenging to identify;
- Can often be prevented by Test Driven Development (TDD)
  - using this strategy we write a test for a feature
  - before actually implementing that feature;
  - initially the test fails (the feature has not yet been implemented),
  - but later can be used to give some confidence on the implementation;

Outline

Debugging

- Debugging is about locating and repairing mistakes in programs.
- Mistakes in programs are usually called *bugs*;
- But repairing a bug is not a *linear* process:
  - when changing a program to fix a bug,
  - there is always the risk that we end up with a program with more bugs!
- If we don't have backups and don't use version control
  - it can be very hard to revert program changes!
- Making regular backups is an *essential part of programming*!

Debugging

Version control systems

- allow us to incrementally save changes at any granularity level,
- and test some changes and revert to an older version if needed;

It is best to check your code into one version control system, before actually starting to debug.

(Believe us,) this will save you precious time in the future!

Debugging

Version control systems

- allow us to incrementally save changes at any granularity level,
- and test some changes and revert to an older version if needed;

It is best to check your code into one version control system, before actually starting to debug.

(Believe us,) this will save you precious time in the future!

Dealing with syntax errors

- If we try to run a program with a syntax error,
  - Python stops execution,
  - and prints:
    - the filename
    - line number
    - offending line, with a caret ^ underneath
- Can you spot the error in the following statement?

```
File "blocks.py", line 383 (or close by)
        if BlockOutput.save_blocks_as_svg(blocks, svg)
                                                    ^

SyntaxError: invalid syntax
```

- We forgot to put a colon (:) at the end of the condition

Dealing with syntax errors

- If we try to run a program with a syntax error,
  - Python stops execution,
  - and prints:
    - the filename
    - line number
    - offending line, with a caret ^ underneath
- Can you spot the error in the following statement?

```
File "blocks.py", line 383 (or close by)
        if BlockOutput.save_blocks_as_svg(blocks, svg)
                                                    ^

SyntaxError: invalid syntax
```

- We forgot to put a colon (:) at the end of the condition

Dealing with syntax errors

- A frequent example which is not at all obvious:

```
File "blocks.py", line 385 (or close by)
        except ValueError as err:
        ^
    SyntaxError: invalid syntax
```

- There is no syntax error in the line indicated!
- So, both the line and the caret's position are wrong.
- In general, when this happens the error will be in an earlier line!
- Looking at the code from the `try` to the `except` clauses:

```
try:
    blocks = parse(blocks)
    svg = file.replace(".blk", ".svg")
    if BlockOutput.save_blocks_as_svg(blocks, svg):
        print("Saved {0}".format(svg))
    else:
        print("Error: failed to save {0}".format(svg)
except ValueError as err:
    print(str(err).format(file))
```

- We notice that the `print()` function's closing parenthesis is missing;

Dealing with syntax errors

- A frequent example which is not at all obvious:

```
File "blocks.py", line 385 (or close by)
        except ValueError as err:
        ^
    SyntaxError: invalid syntax
```

- There is no syntax error in the line indicated!
- So, both the line and the caret's position are wrong.
- In general, when this happens the error will be in an earlier line!
- Looking at the code from the `try` to the `except` clauses:

```
try:
    blocks = parse(blocks)
    svg = file.replace(".blk", ".svg")
    if BlockOutput.save_blocks_as_svg(blocks, svg):
        print("Saved {0}".format(svg))
    else:
        print("Error: failed to save {0}".format(svg)
except ValueError as err:
    print(str(err).format(file))
```

- We notice that the `print()` function's closing parenthesis is missing;

Dealing with runtime errors

- If an unhandled exception occurs at runtime,
  - Python stops execution
  - and prints a traceback

```
Traceback (most recent call last):
  File "blocks.py", line 392, in <module>
    main()
  File "blocks.py", line 381, in main
    blocks = parse(blocks)
  File "blocks.py", line 174, in parse
    return data.stack[1]
IndexError: list index out of range
```

Dealing with runtime errors

- Tracebacks should be read from their last line back to their first line.

```
Traceback (most recent call last):
  File "blocks.py", line 392, in <module>
    main()
  File "blocks.py", line 381, in main
    blocks = parse(blocks)
  File "blocks.py", line 174, in parse
    return data.stack[1]
IndexError: list index out of range
```

- The last line specifies the unhandled exception that occurred;
- Above this line, Python shows:
  - the filename, line number and function name, followed,
  - in a new line, by the line that caused the exception
- If the function where the exception was raised was called by another function, the elements above are shown for the calling function;
- This goes all the way up to the beginning of the call stack.

Dealing with runtime errors

- In this case:

```
Traceback (most recent call last):
  File "blocks.py", line 392, in <module>
    main()
  File "blocks.py", line 381, in main
    blocks = parse(blocks)
  File "blocks.py", line 174, in parse
    return data.stack[1]
IndexError: list index out of range
```

- `data.stack` is a (type of) sequence which has no item at position 1;
- The error occurred at line 174 of `blocks.py`'s `parse()` function;
- `parse()` was called at line 381 in the `main()` function;
- The call to `main()` was made at line 392,
  - and this is the line at which program execution started.

Dealing with runtime errors

- Tracebacks may look intimidating
- But having understood them, they are easy to follow and quite useful
- In any case, they *only* tell us where to look for the problem,
- And not how to solve it!

Dealing with runtime errors

- Let's look at another interesting example:

```
Traceback (most recent call last):
  File "blocks.py", line 392, in <module>
    main()
  File "blocks.py", line 383, in main
    if BlockOutput.save_blocks_as_svg(blocks, svg):
  File "BlockOutput.py", line 141, in save_blocks_as_svg
    widths, rows = compute_widths_and_rows(cells, SCALE_BY)
  File "BlockOutput.py", line 95, in compute_widths_and_rows
    width = len(cell.text) // cell.columns
ZeroDivisionError: integer division or modulo by zero
```

- The problem was in `BlockOutput.py`, that is called by `blocks.py`
- I.e., the traceback tells us where the problem became *apparent*, not where it *occurred*!
- The value of `cell.columns` is clearly 0 in `BlockOutput.py`'s `compute_widths_and_rows()` function on line 95
- But we have to analyze the program to find where/why this happened!

Dealing with runtime errors

- The traceback sometimes reveals an exception that occurred
  - in Python's standard library, or in a third-party library
- This might reveal a bug in the library, but most likely it is on our code!

```
Traceback (most recent call last):
  File "blocks.py", line 392, in <module>
    main()
  File "blocks.py", line 379, in main
    blocks = open(file, encoding="utf8").read()
  File "/usr/lib/python3.0/lib/python3.0/io.py", line 278, in __new__
    return open(*args, **kwargs)
  File "/usr/lib/python3.0/lib/python3.0/io.py", line 222, in open
    closefd)
  File "/usr/lib/python3.0/lib/python3.0/io.py", line 619, in __init__
    _fileio._FileIO.__init__(self, name, mode, closefd)
IOError: [Errno 2] No such file or directory: 'hierarchy.blk'
```

- The `IOError` tells us clearly what the problem is!
- But the exception was raised in the standard library's `io` module
- In such cases, we should keep reading upward until our own first file

Dealing with runtime errors

- The first reference is to `blocks.py`, line 379, in `main()`

```
Traceback (most recent call last):
  File "blocks.py", line 392, in <module>
    main()
  File "blocks.py", line 379, in main
    blocks = open(file, encoding="utf8").read()
  File "/usr/lib/python3.0/lib/python3.0/io.py", line 278, in __new__
    return open(*args, **kwargs)
  File "/usr/lib/python3.0/lib/python3.0/io.py", line 222, in open
    closefd)
  File "/usr/lib/python3.0/lib/python3.0/io.py", line 619, in __init__
    _fileio._FileIO.__init__(self, name, mode, closefd)
IOError: [Errno 2] No such file or directory: 'hierarchy.blk'
```

- A call being made to `open()` is erroneous
- So we may have forgotten to put it inside a `try ... except` block
- We should modify `blocks.py` to cope gracefully when dealing with nonexistent files
- This is a usability error, and should be classified as a logical error!

Dealing with runtime errors

NOTE: if you are using, e.g., Python 3.1 you may already see shorther (and smarter) tracebacks!

- We should always catch all relevant exceptions
  - It is not acceptable to show a traceback to a user!
- Exceptions that we catch and cannot recover from should be reported in the form of error messages
- The same applies for programs with a graphical interface, except in this case we should use a message box to notify the user of a problem

Scientific Debugging

- If our program runs but does not have the expected/desired behavior
  - we have a bug - a logical error - that we must eliminate
- We suggest to use TDD in a systematic way
  - to prevent the occurrence of errors in the first place
- But some bugs will always get through
- Debugging is a necessary skill to learn!

Scientific Debugging

- We will describe a systematic approach to debugging
- It is detailed and might suggest too much work to handle
- But following the process will avoid wasting time with *random* debugging

To be able to kill a bug, we must be able to do the following:

1. Reproduce the bug
2. Locate the bug
3. Fix the bug
4. Test the fix

1. Reproducing the bug

- Reproducing the bug is sometimes easy
  - if it occurs on every run, for example
- In other occasions, this might be difficult to achieve
  - if it occurs on some runs but not on others, for example

In either case, we should find the smallest input and the least amount of processing that can still produce the bug.

2-3. Finding and Fixing the bug

- Once we were able to reproduce the bug, we have
  - the input data and options, and
  - the incorrect results
- With this information, we can apply the scientific method to finding and fixing the bug;
- The method has 3 steps:
  1. Think of an explanation - a hypothesis - that reasonably accounts for the bug
  2. Create an experiment to test the hypothesis
  3. Run the experiment
- Running the experiment should help to locate the bug

  while also giving some insight into its solution

2-3. Finding and Fixing the bug

1. Think of an explanation - a hypothesis - that reasonably accounts for the bug
   - E.g., do we suspect of a particular function?
2. Create an experiment to test the hypothesis
   - We should check the arguments the function receives, the values of its local variables and the return value
3. Run the experiment
   - We run the program with data we know produces errors
   - If the arguments coming into the function are not the ones expected,
     - the problem must be further up the call stack
     - we should repeat the method suspecting of a function that calls the one we have looked at
   - If all incoming arguments are always valid, we must look at local variables and return value
   - It these are always correct, we must come up with a new hypothesis
     - The suspect function is behaving correctly
   - If not, we know we need to investigate the function further

2-3. Finding and Fixing the bug

- To investigate a function further
  - We can use an intrusive approach
    - By inserting `print()` statements in the program
  - Or we can use a non-intrusive approach
    - By using a debugger
  - Both are used and are valid

2-3. Finding and Fixing the bug

The intrusive approach

- We can start by putting a `print()` statement right at the beginning of the function
  - so that it prints the function's arguments
- Then, just before the (or each) `return` statement
  - add `print(locals(), "\n")`
    - `locals()` returns a dictionary whose keys are the names of the local variables and whose values are the variables' values
  - we can simply print the values of some variables, if preferable
- If we see that the arguments are correct but the return value is in error, we know the problem is in the function
- If this does not immediately occur, we can add `print()` calls in more places to try to further isolate the statements of the function

2-3. Finding and Fixing the bug

The non-intrusive approach

- The alternative is to use a debugger, e.g., the `pdb` module
  - add `import pdb` in the program itself
  - add `pdb.set_trace()` as the 1st statement of the function to examine
  - when the program is run, `pdb` stops it immediately after the `pdb.set_trace()` call
  - and allows us to step through the program, set breakpoints and examine variables

2-3. Finding and Fixing the bug

The non-intrusive approach

- Inspecting the `calculate_median()` function of `statistics.py`:

```
python statistics.py sum.dat
> statistics.py(76)calculate_median()
-> numbers = sorted(numbers)
(Pdb) p numbers
[7.0, 33.0, 50.0, 88.0, 2.0, 91.0, 20.0, 93.0, 60.0, 80.0, 43.0, 93.0, 67.0, 8
(Pdb) s
> statistics.py(77)calculate_median()
-> middle = len(numbers) // 2
(Pdb) p numbers
[2.0, 3.0, 7.0, 11.0, 11.0, 14.0, 15.0, 20.0, 21.0, 24.0, 26.0, 26.0, 28.0, 33
(Pdb) s
> statistics.py(78)calculate_median()
-> median = numbers[middle]
(Pdb) p middle
18
(Pdb) c
count      =     37
mean       =     49.70
median     =     50.00
std. dev.  =     30.28
```

2-3. Finding and Fixing the bug

The non-intrusive approach

- The alternative is to use a debugger, e.g., the `pdb` module
  - commands are written after the `(Pdb)` prompt
  - `s` means step, i.e., execute the statement shown
  - `p` means print, we have used it, e.g., to show the value of `numbers`
  - `c` means continue to the end of the program

4. Test the fix

- Once the fix is in place, we must test it
- This includes
  - testing to see if the bug it is intended to fix has gone away
  - but also that our fix has not introduced other bugs
- So we must test the bugfix, and *all* the program's tests to increase our confidence that the bug did not have unwanted effects

Outline

Unit Testing

- Testing can **not** guarantee the correctness of a (nontrivial) program
  - the range of possible (valid and invalid) inputs
  - and the range of possible computations is too vast!
- But by carefully choosing what we test, we can improve code!
- Different kinds of tests exist, such as functional or usability tests
- We will focus on unit testing
  - which consists of testing individual functions, classes and methods

Unit Testing

- TDD conveys that we start by writing test(s) for a method
- And only then we implement the method
- Finally, we run the test(s) for that method
- As well as all we rerun the tests for all methods to make sure our addition hasn't had any unexpected side effects
- Once all tests pass, we can be reasonably confident that our program does what is expected

Unit Testing

- Let's build a function to insert a string at a particular index position
- We start by writing its tests using `doctests`:

```
def insert_at(string, position, insert):
    """Returns a copy of string with insert inserted at the position

    >>> string = "ABCDE"
    >>> result = []
    >>> for i in range(-2, len(string) + 2):
    ...     result.append(insert_at(string, i, "-"))
    >>> result[:5]
    ['ABC-DE', 'ABCD-E', '-ABCDE', 'A-BCDE', 'AB-CDE']
    >>> result[5:]
    ['ABC-DE', 'ABCD-E', 'ABCDE-', 'ABCDE-']
    """
    return string
```

- For now, the function returns the unchanged value of input `string`

Unit Testing

- So both tests fail:

```
> python -m doctest my_string.py
**********************************************************************
File "my_string.py", line 8, in my_string.insert_at
Failed example:
    result[:5]
Expected:
    ['ABC-DE', 'ABCD-E', '-ABCDE', 'A-BCDE', 'AB-CDE']
Got:
    ['ABCDE', 'ABCDE', 'ABCDE', 'ABCDE', 'ABCDE']
**********************************************************************
File "my_string.py", line 10, in my_string.insert_at
Failed example:
    result[5:]
Expected:
    ['ABC-DE', 'ABCD-E', 'ABCDE-', 'ABCDE-']
Got:
    ['ABCDE', 'ABCDE', 'ABCDE', 'ABCDE']
**********************************************************************
1 items had failures:
   2 of   5 in my_string.insert_at
***Test Failed*** 2 failures.
```

Unit Testing

- But if we correctly implement the function:

```
def insert_at(string, position, insert):
    """Returns a copy of string with insert inserted at the position

    >>> string = "ABCDE"
    >>> result = []
    >>> for i in range(-2, len(string) + 2):
    ...     result.append(insert_at(string, i, "-"))
    >>> result[:5]
    ['ABC-DE', 'ABCD-E', '-ABCDE', 'A-BCDE', 'AB-CDE']
    >>> result[5:]
    ['ABC-DE', 'ABCD-E', 'ABCDE-', 'ABCDE-']
    """
    return string[:position] + insert + string[position:]
```

```
> python -m doctest my_string.py
>
```

- No output being provided means all tests passed!

Unit Testing

- Creating `doctests` is easy
  - we write the tests in the module, function, class and method `docstring`
- Then, for modules, we simply add to the end of the module:

```
if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

- For programs, even if we say that we want to execute the `main()` function:

```
if __name__ == "__main__":
    main()
```

- We still can execute the program's doctests by doing

```
> python -m doctest _program_.py
```

- where `_program_` should be replaced by the name of the program

Outline

Profiling

- Even a program that produces the expected results may run slowly
  - The program may also over-spend other resources such as memory
- Ideally, programs should be optimized to spend minimal resources
- We should find out precisely where performance issues may be
- Randomly optimizing can cause us to introduce bugs or to speed up parts of our program that have no effect on the program's overall performance.

Profiling

There are a few Python habits good for performance:

1. Prefer tuples over lists when a read-only sequence is needed
2. Use generators rather than creating large tuples or lists to iterate over
3. Use Python's built-in data structures, e.g., `dicts`, `lists` and `tuples`
   - rather than custom data structures
4. When creating large strings out of small strings, instead of concatenating the small strings,
   - accumulate them in a list, and join them only once
5. If an object (including a function/method) is accessed many times (e.g., when accessing a function in a module), or from a data structure,
   - it may be better to use a local variable that refers to the object
   - this provides faster access

Profiling

- Even following the guidelines, you may define performance bottlenecks
  - This might be due, e.g., to poor choice of algorithms or data structures
- We will study the use of the `timeit` module for profiling
- Imagine we have three functions, `function_a()`, `function_b()` and `function_c()`
  - all performing the same computation
  - but each using a different algorithm
- We can run and compare these functions using the `timeit` module

Profiling

- We, e.g., place them into a module, and add the code to the end of the module:

```
if __name__ == "__main__":
    repeats = 1000
    for function in ("function_a", "function_b", "function_c"):
        t = timeit.Timer("{0}(X, Y)".format(function),
                         "from __main__ import {0}, X, Y".format(function))
        sec = t.timeit(repeats) / repeats
        print("{function}() {sec:.6f} sec".format(**locals()))
```

- `timeit.Timer()` receives two arguments:
  1. a string with the code we want to execute
     - in the first iteration of the loop, this would be `function_a(X,Y)`
  2. is optional; if present, it is a string with code to be executed *before* the code to be timed
     - here, we have just imported from the `__main__` module (i.e., this module) the function we want to test, plus variables `X` and `Y` which are global at the same module

Profiling

- When the `timeit.Timer` object's `timeit()` method is called
  1. it will first execute the constructor's second argument (if it exists)
  2. it will then execute the constructor's first argument
     - **and time how long the execution takes**
- The `timeit.Timer.timeit()` method's return value is the time taken in seconds, as a float
- By default, the method repeats 1 million times
  - but in this case, we have needed only 1000 repeats to observe a difference
- Finally, we calculate the average execution time, and print the name and execution time on the console

```
function_a() 0.001618 sec
function_b() 0.012786 sec
function_c() 0.003248 sec
```

- With the input data that was used, `function_a()` is the fastest
  - But we must make sure the data we use is representative!