CS 61A SU26

Lecture 11: Exceptions II

July 8, 2026

Rebecca Dang

Join pollev.com/rebeccadang831 (or scan QR code) on your phone or laptop

We'll begin at Berkeley time (10 minutes after), as per tradition!

Potpourri

10 min

- Announcements
- Midterm Advice
- Midterm Review Survey

Announcements

- **Midterm exam is Mon 7/13 from 7-9 pm**: Ed logistics post has been released
  - As always, it is your responsibility to read through the whole post! Some highlights below
- In addition to the provided midterm study guide/reference sheet we will provide, **you can bring your own notes sheet** (1 sheet of US letter paper, handwritten, can use both sides)
- **Scope:** All topics through lecture tomorrow (Midterm Review)
- Check the post for what is **cancelled** the day of and the day after the midterm (e.g. no lecture on 7/13)
- **[Action Required]** If you need to request an alternate time, have extended time accommodations, want to request a left-handed desk, etc. **submit the Midterm Alteration Form by Fri 7/10 at 11:59 pm PT**
- Seating and alteration **confirmation emails sent Sun 7/12 evening**
- Extenuating circumstances? Email cs61a@berkeley.edu and refer to the syllabus policies on exams
- Quiz 2 grades released today or tomorrow

Midterm Advice (1 of 3)

- Complete all assignments (on your own) that are due by tomorrow
- If you missed a lecture, watch the recording and actually do the PollEv questions and coding practice as if you were there
- If you haven't already, go over your quizzes in OH (even if you got a high score)
- Attend the Midterm Review Session on Fri 7/10 from 3-5 PM in HP Auditorium (Soda 306) or watch the recording
- Do past exams (active > passive learning)
  - **Do at least 1-2 past midterm exams in their entirety**
    - I recommend the most recent summer midterms since their scope will be more closely aligned to ours (but not guaranteed).
  - Once you have identified the topics you struggle with most, **concentrate on doing practice problems for those topics**
  - See also: Exam prep worksheets

Midterm Advice (2 of 3)

What do you mean by "doing problems"?

- **Replicate exam conditions:** Handwrite your answers, no assistance aside from the reference sheet and your notes sheet, time yourself
- **Go over the solutions:** Read the solutions PDF and make sure you understand *why* the solution is correct. If you're confused, watch the walkthrough video (if available) and/or make an Ed post and/or come to OH!
- **Quality > quantity. Focus on learning the actual content**, not scoring points or doing as many problems as you can.
- **Remember test-taking strategies:** You don't have to complete problems in order, you might be able to use process of elimination, and guessing is better than leaving it blank

Midterm Advice (3 of 3)

- **Be careful using LLMs to study:** They may hallucinate, give you out of scope or incorrect information, etc. That said, they can be useful resources for explaining things in a simpler way or generating practice problems, as long as you double check that it's correct.
- From past students: Advice from Former Students
- From staff:
  - Preparation section of the Midterm Logistics Post
  - How to Do Well on the Midterm
  - Midterm Tips + Commonly Missed Points
- From my personal experience taking 61A in FA21 (could be outdated)
- **Remember: No matter what happens, you are not your grades and the course is not curved!**
  - Exam averages for a UC Berkeley CS course are typically ~65%, average grade is a B+

Midterm Review Survey

PollEv

Demo Repository

1. Go to https://github.com/phrdang/cs61a-su26-lec11
2. Click on the green "Code" dropdown in the top right
3. Click "Download Zip"
4. Download the .zip file of the repository wherever you wish (recommended: your existing `cs61a` directory)
5. Unzip the file
6. Open the `cs61a-su26-lec11` directory in VS Code
7. Follow the instructions in `README.md` under "Installation" to get set up
8. Follow along with demos and exercises today

Exceptions

5 min

- Recall: Exceptions
- Demo: Debugging Exercise

Recall: Exceptions

Exceptions are errors that are raised/thrown during the execution of a program when non-standard behavior occurs

Demo: Debugging Exercise

- Download `11-debug.py` from the course website
- Follow along as we debug this file!
- Solution is in `11-debug-sol.py`

Defensive Programming

10 min

- Recall: `assert` and type hints
- Common Python Exceptions
- `raise`
- Problem Decomposition
- Code Style/Quality

Recall: `assert` and type hints (1 of 3)

- Recall that defensive programming refers to techniques to prevent bugs in code
- Recall: We can use `assert <condition>, <message>` statements to explicitly check if our assumptions are true, and raise an `AssertionError` if the condition is falsy
- Recall: We can use type hints to tell other programmers what the data types of our function inputs/outputs and variables are. Ex:

```
def add(a: int, b: int) -> int:
    return a + b
```

Recall: `assert` and type hints (2 of 3)

Now that we've learned about containers, note that you can also use type hints for them! Ex:

```
def to_string(lst: list[str]) -> str:
    total = ""
    for s in lst:
        total += s
    return total
```

Recall: `assert` and type hints (3 of 3)

- Note: Because Python is technically a dynamically typed language, **it does not actually enforce type hints at runtime** (Python docs).
- However, type hints can be used by static analysis tools like linters
  - **Static analysis tool** = Tool that analyzes written code without executing it
  - **Linter** = Type of static analysis tool that gives feedback on code "best practices"
    - Ex: Ruff, Pylint
    - Fun fact: You can install extensions for most major linters in VS Code to lint your Python code!

Common Python Exceptions (1 of 3)

Exception	What it usually means
`AssertionError`	A programmer-defined precondition for the program to execute is not true.
`TypeError`	Raised when an operation or function is applied to an object of inappropriate type
`ValueError`	Raised when an operation or function receives an argument that has the right type but an inappropriate value, and the situation is not described by a more precise exception such as `IndexError`.
`SyntaxError`	There is something wrong with the "form" of your program, e.g. missing parentheses, brackets, colons, indentation issues, invalid expressions, misspelled keywords, etc.
`RecursionError`	Infinite recursion / stack overflow

Common Python Exceptions (2 of 3)

Exception	What it usually means
`KeyError`	Tried to access a value at a particular key in a dictionary, but the key doesn't exist
`NameError`	Tried to access a name (e.g. function or variable) that doesn't exist. Often caused by typos.
`IndexError`	Raised when a sequence index is out of range.
`StopIteration`	Raised when an iterator runs out of elements.
`UnboundLocalError`	Raised when a reference is made to a local variable in a function or method, but no value has been bound to that variable.

Common Python Exceptions (3 of 3)

Exception	What it usually means
`ZeroDivisionError`	Division by zero
`RuntimeError`	Raised when an error is detected that doesn't fall in any of the other categories.
`AttributeError`	Raised when an attribute reference or assignment fails (more on this when we talk about OOP!)

raise (1 of 2)

Instead of generic `AssertionError`s, if we want to be more descriptive about the unexpected behavior that occurred, we can `raise` our own exceptions!

Syntax: `raise <exception>` where `<exception>` is either an exception class or instance (e.g. if we want to pass in a message).

Examples:

```
raise TypeError
raise ValueError('n must be a positive integer')
```

Note: Why raise an exception? Can't we just do nothing?

raise (2 of 2)

```
def square_root(x: float) -> float:
    if x < 0:
        raise ValueError('x must be non-negative')
    return x ** 0.5
```

Decomposition

- In 61A, you will typically implement single functions or small projects
- In real world applications, you might have hundreds or thousands of code files with millions of lines of code in a single codebase
- Thus, it is crucial to **decompose** problems into subproblems, and **separate concerns** of different parts of the code to **avoid repetition**
  - Ex: Creating helper functions
- Some linters can perform **cyclomatic complexity analysis**, which returns a number corresponding to how "complex" your functions are so you know when it's a good time to separate things out or reduce repetition
  - Demo: `ruff check complexity.py`

Code Style/Quality

- Although code quality has many definitions, we define **code style defect** as "**a part of code that works correctly but could be written in a clearer, more elegant, or more conventional way**" (Řechtáčková et al. 2025)
- Why do we care? Code is written not just for machines, but also fellow humans. **Readable code is easier to maintain and debug!**
- Linters and **formatters** (static analysis tools that automatically fix certain style defects) can help us improve our code style!
  - Ex: Ruff (it's both a linter and formatter), Black
- Demo: `ruff format format.py`

Test-Driven Development

30 min

- Definition
- Types of Tests
- Testing Frameworks
- Test Coverage
- Exercise: Count Vowel Occurrences
- Exercise: Grading Analytics

Definition

Test-driven development (TDD) is a software engineering practice where you:

1. Read/write your software specification (e.g. what it should do)
2. Write tests about how you expect your software to behave under certain conditions
3. Write code
4. Run the tests
5. Repeat steps 2-4 as needed until all tests pass

Why write tests?

- To automate the process of **software verification**, especially for large, interconnected codebases where multiple people are working simultaneously (ex: Stripe Ruby Infra blog post)
- To **catch edge cases** before they happen and ensure they're handled

Types of Tests

- **Manual test:** Run the program and call the function yourself to manually check it works
- **Unit test:** Code that checks if an isolated, small part of the code (e.g. a single function) works correctly
  - Ex: Python Doctests
- **Integration test:** Code that checks if multiple parts of the code work when combined together
- **End-to-end (E2E) test**: Code that checks if the entire system works

Testing Frameworks

- Most programming languages have **testing frameworks** which make it easy to write and run tests
- For example, the `doctest` module is an example built-in to Python
- Another popular Python testing framework is `pytest`, which can be installed separately for more heavy-duty applications

Test Coverage

- Many testing frameworks also provide a statistic called **test coverage**, which is the percentage of lines of code that are executed when running a suite of tests
- **Higher coverage is generally better**, because that means that your source code is being tested in all/most of its possible paths of execution
  - 100% coverage is rare (and perhaps undesirable) in practice because code is constantly changing and because you don't want to "overfit" your tests
  - 100% coverage also doesn't necessarily mean **all** execution paths are accounted for, but it means at some point all the lines were executed in the tests

Exercise: Count Vowel Occurrences (10 min)

1. In the root directory of the repository, you can create a test coverage report by running the command in Line 2 of `count_vowel_occurrences.py`
    a. Open the report by opening the `htmlcov/index.html` file in your browser (demo)
    b. Refresh the page in your browser to see the updated report after rerunning the coverage command
2. Based on the output, write a single doctest that results in 100% coverage (see existing ones for examples)

Challenge: Even though you can reach 100% coverage with one doctest, the tests aren't actually comprehensive of the function's behavior. Write more doctests to make sure it works properly! The solution is in `count_vowel_occurrences_sol.py`

Exercise: Grading Analytics (10 min)

1. In the root directory of the repository, you can run tests with `uv run pytest tests/test_grade_analytics.py`. You should see that there are 3 test failures.
2. Based on the output, fix the source code in `src/grade_analytics.py`
   a. The tests are defined in `tests/test_grade_analytics.py`
   b. Don't peek at the solution in `src/grade_analytics_sol.py` until you're done (search for `# FIX` for the fixes)!

5 min break

Exception Handling

5 min

- `try` statements

`try` statement (1 of 3)

To handle exceptions gracefully in Python, you can use `try` statements:

```
try:
    <try suite>
except <exception> as <name>:  # optional, can have multiple
    <except suite>
else:  # optional
    <else suite>
finally:  # optional
    <finally suite>
```

Note 1: At minimum, you need `try` with `except`, or `try` with `finally`

Note 2: If you only write `except:` without an exception class, it will catch any exception. This can be dangerous (remember, we don't want to fail silently), so it's best practice to always specify!

try statement (2 of 3)

```
def safe_divide(numerator, denominator):
    try:
        result = numerator / denominator
    except ZeroDivisionError:
        result = 'Undefined'
    else:
        print(f"The result of the division is: {result}")
    finally:
        print("Cleanup")
    return result
```

try statement (3 of 3)

```
>>> 1 / 0
Traceback (most recent call last):
  File "<python-input-0>", line 1, in <module>
    1 / 0
    ~~^~~
ZeroDivisionError: division by zero
>>> safe_divide(1, 0)
Cleanup
'Undefined'
>>> safe_divide(4, 2)
The result of the division is: 2.0
Cleanup
2.0
```

Debugging Methods

15 min

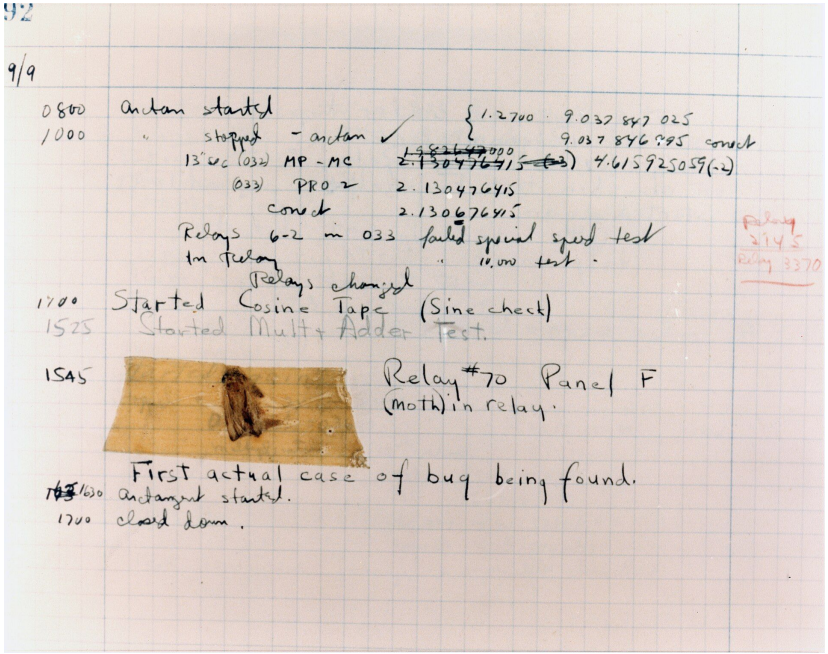
- Where does "bug" come from?
- General Debugging Approach
- Advanced Debugging Methods
- Exercise: Print Debugging

Where does "bug" come from?

"The term bug to describe a defect **has been engineering jargon since at least as far back as the 1870s**, long before electronic computers and computer software....

While [computer pioneer Grace] Hopper was working on the Mark II and Mark III as Harvard faculty in about 1947, **operators traced an error in the Mark II to a moth trapped in a relay**. The moth was removed from the mechanism and taped in a log book with the note 'First actual case of bug being found.'"

- Wikipedia



General Debugging Approach

1. Read the problem (and any provided tests) carefully to understand the expected behavior
2. Run the tests for the code you wrote
3. Read the test output carefully: What's the difference between the expected and actual value(s) returned or displayed?
4. Make a hypothesis based on step 3: Why might this have happened?
5. Test the hypothesis

This lecture intentionally does not cover AI-assisted tools for coding, we might talk about that later in a special topics lecture though!

Advanced Debugging Methods

- `python3 -i code.py` and manually call your function to test it
- Draw an environment diagram or put your code through Python Tutor (make sure to actually call the function)
- pdb module to add *breakpoints* (places where you pause execution of the program to inspect the current state) as code
- Debuggers, e.g. VS Code Python debugger
  - Graphical user interface to add breakpoints and view current state
  - In CS 61B you'll learn the IntelliJ debugger (for Java)
  - In CS 61C, 161, and 162 you'll learn the `gdb` debugger (for the C programming language)

Exercise: Print Debugging (10 min)

1. Read the contents of the source file `src/make_acronym.py`
2. In the root directory of the repository, run `uv run pytest tests/test_make_acronym.py`
3. Use print debugging to determine the bug(s) and fix them
4. Solution in `make_acronym_sol.py`

Hint: Google is your friend for unfamiliar methods or to search up how to do XYZ in Python!