

Try / Except Statements

Sometimes you need to run some code that might cause an error for reasons outside your control. A `try / except` statement lets you catch and deal with errors.

try:

<code you need to run>

except <error type>:

<what to do if there's an error>

```
num = input("Enter a number: ")
try:
    num = int(num)
    print("Square:", num**2)
except ValueError:
    print("That's not a number!")
```

Debugging

15-110 – Monday 09/21

Announcements

- Hw4 was due today
- Hw5 not due until **Mon 10/05**, will be released soon
 - You get a week off to study for the exam
- Hw3 revisions due **tomorrow noon**
- Quizlet3 on Wednesday
- Don't forget to sign up for an Oral1 timeslot!
 - Students who don't complete this assessment automatically get a 0 on one of Hw2/Hw3

Announcements – Exam1

- **Exam1 is next Wednesday 09/30**
 - Will cover content from **Weeks 1-4**
- Exam problems are similar to homework problems, but all solved on paper.
- You have the whole lecture period (50 minutes) to take the exam.
- **Different location than usual!!**
 - Lecture 1 (2pm): CUC McConomy
 - Lecture 2 (4pm): DH 2315

Announcements – Exam1

- **Exam logistics:**
 - **Open-note** – you may bring up to five pages of paper notes
 - Front and back is fine – five total papers
 - **Closed to collaboration** – don't discuss the exam with others until feedback is released
 - **What to bring** - your notes, something to write with, and your andrewID card (we'll check IDs at the door)

Announcements – Exam1

- Exam Review Materials: <https://www.cs.cmu.edu/~110/assessments.html>
 - **Topics List:** knowledge components that may be tested
 - **Practice Exam:** a variant exam designed by course TAs
 - May or may not be the same difficulty level as the exam, but should give you an idea of what the exam looks like
 - **Extra Problems:** practice problems generated by the TAs
 - **Knowledge Tree:** practice content created by researchers outside the course
- **TA Review Session:** content & example review. **Sunday 09/27 2:30pm, GHC 4401**
- **TA Small Groups:** contact your TA for times

Soon!



Learning Goals

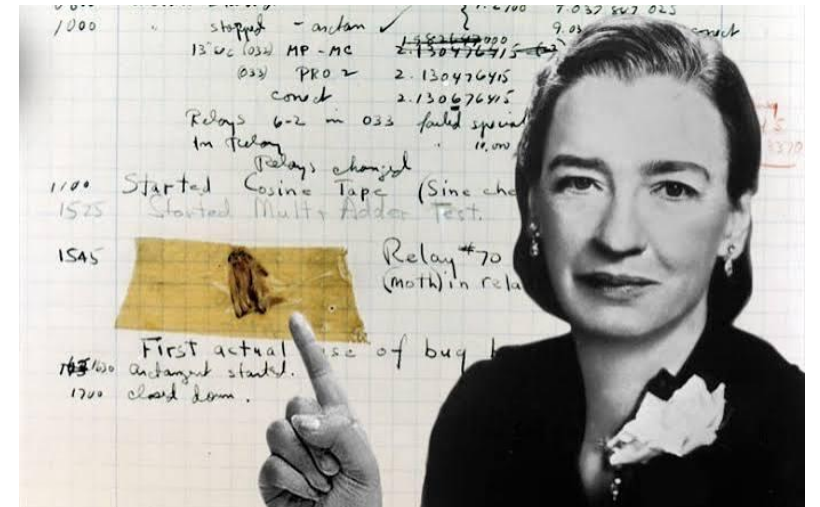
- Interpret **error messages** generated by Python programs.
- Use **debugging** on a Python program that is not working correctly to identify where the problem occurs and fix simple bugs.

Where does the term "bug" come from?

Thomas Edison, 1878, in a letter about his telegraph system talked about chasing down small faults he called "bugs".

Grace Hopper, 1947, found an actual moth stuck in a Harvard Mark II computer, blocking an electrical contact.

Hopper taped it into the logbook and wrote:
"First actual case of bug being found"



Why do bugs happen?

Humans make mistakes. We forget a step, mistype something, misremember how something works. Nobody writes perfect code, professionals included.

AI can make mistakes. As we use AI to write code, it can introduce errors unintentionally. AI-written code is not automatically correct, we still need to be able to read it, run it, and check it.

Debugging isn't a sign you did something wrong. It's just part of programming.

Reminder: Types of Errors

There are three types of errors that we may encounter while coding.

Syntax Error: Python was not able to parse the code

Runtime Error: Something went wrong when the code was running

Logical Error: The code ran, but didn't do what we wanted it to do

Syntax Errors

Syntax Error Example

There is an error in the code below. Can you find it?

```
1 def calculate_average(numbers)
2     total = sum(numbers)
3     count = len(numbers)
4     average = total / count
5     return average
6
```

Let's try running the code.

Reminder: Syntax Error Debugging Process

1. Look for the **line number**. This line tells you approximately where the error occurred.
2. Look at the **error type**.
3. If it says **SyntaxError**, look for the **inline arrow**. The position gives you more information about the location of the problem (though it isn't always right).
4. If it says something else, **read the error message**. The error type and its message give you information about what went wrong.

```
>>> %Run -c $EDITOR_CONTENT
Traceback (most recent call last):
  File "<string>", line 1
    def calculate_average(grades)
                          ^
SyntaxError: expected ':'
```

Annotations:

- line number: points to `line 1`
- inline arrow: points to `^`
- error type: points to `SyntaxError`

Careful: The inline arrow might not point to where the bug is

```
1 def print_report(name, average):  
2     """  
3     Prints a formatted report for the student  
4     print(name, average)  
5  
6 def letter_grade(average):  
7     if average >= 90:  
8         return "A"  
9     return "F"
```

```
>>> %Run -c $EDITOR_CONTENT  
Traceback (most recent call last):  
  File "<string>", line 2  
    """  
    ^  
SyntaxError: unterminated triple-quoted string literal (detected at line 10)
```



Common Causes of Syntax Errors

Missing or misplaced punctuation: Incorrect use of parentheses, brackets, or quotes.

Incorrect indentation (`IndentationError`, `TabError`): Python relies heavily on indentation to define code blocks. Incorrect indentation can lead to syntax errors.

Invalid keywords or identifiers: Using reserved keywords as variable names.

Misspelling: Python is a case-sensitive programming language. Misspelling variable names or data types, even with uppercase or lowercase letters.

Runtime Errors

Runtime Error Example

We fixed the syntax error from before. There is another error in the code below. Can you find it?

```
1 def calculate_average(numbers):  
2     total = sum(numbers)  
3     count = len(numbers)  
4     average = total / count  
5     return average  
6  
7  
8 calculate_average([1, 2, 3])  
9 calculate_average([])
```

Note: To get runtime errors we **must** call the function with some input!



Let's run it in Thonny!

```
1 def calculate_average(numbers):  
2     total = sum(numbers)  
3     count = len(numbers)  
4     average = total / count  
5     return average  
6  
7  
8 calculate_average([1,2,3])  
9 calculate_average([])
```

```
>>> %Run -c $EDITOR_CONTENT
```

```
Traceback (most recent call last):
```

```
  File "<string>", line 9, in <module>
```

```
  File "<string>", line 4, in calculate_average
```

```
ZeroDivisionError: division by zero
```

} Function
call
tracing!

Using Print Statements

As Python is running a program, we can't see inside it. A **print statement** is a window into what variables actually hold at that exact moment.

Print statements let you:

- confirm a value is what you think it is
- confirm a line of code was reached (i.e., did we enter the loop?)
- see how far the program got before it crashed

Intentional Printing

Label your prints so you can tell them apart:

`print("count:", count)` instead of `print(count)`

Place a print right *before* the line that crashes or looks suspicious.

If you're not sure which of several lines is the problem, print between each one to watch the program's progress line by line.

Once the bug is fixed, delete the print statements (or comment them out); they're scaffolding, not part of the final program.

Apply to our Runtime Error Code

```
1 def calculate_average(numbers):
2     total = sum(numbers)
3     count = len(numbers)
4     print("#1: Value of total:", total)
5     print("#2: Value of count:", count)
6
7     average = total / count
8     print("#3: Value of average before return:", total)
9     print("-----")
10
11     return average
12
13
14 calculate_average([1,2,3])
15 calculate_average([])
```

```
>>> %Run -c $EDITOR_CONTENT
#1: Value of total: 6
#2: Value of count: 3
#3: Value of average before return: 6
-----
#1: Value of total: 0
#2: Value of count: 0
Traceback (most recent call last):
  File "<string>", line 15, in <module>
  File "<string>", line 7, in calculate_average
ZeroDivisionError: division by zero
```

Common Runtime Errors

IndexError: accessing a list (or string) position that doesn't exist

TypeError: mixing incompatible types (i.e., can't add a string and a number)

NameError : using a variable that was never defined (often a typo)

But there's many other types of runtime errors too...

When Encountering New Errors...

What to do when you don't recognize an error?

1. Read the error type name and message first; you can often derive meaning from that.
2. Copy the exact error type into a search engine, e.g. "*Python TypeError unsupported operand type(s)*".
3. Check the official Python docs or other online resources; they almost always cover your exact situation already.

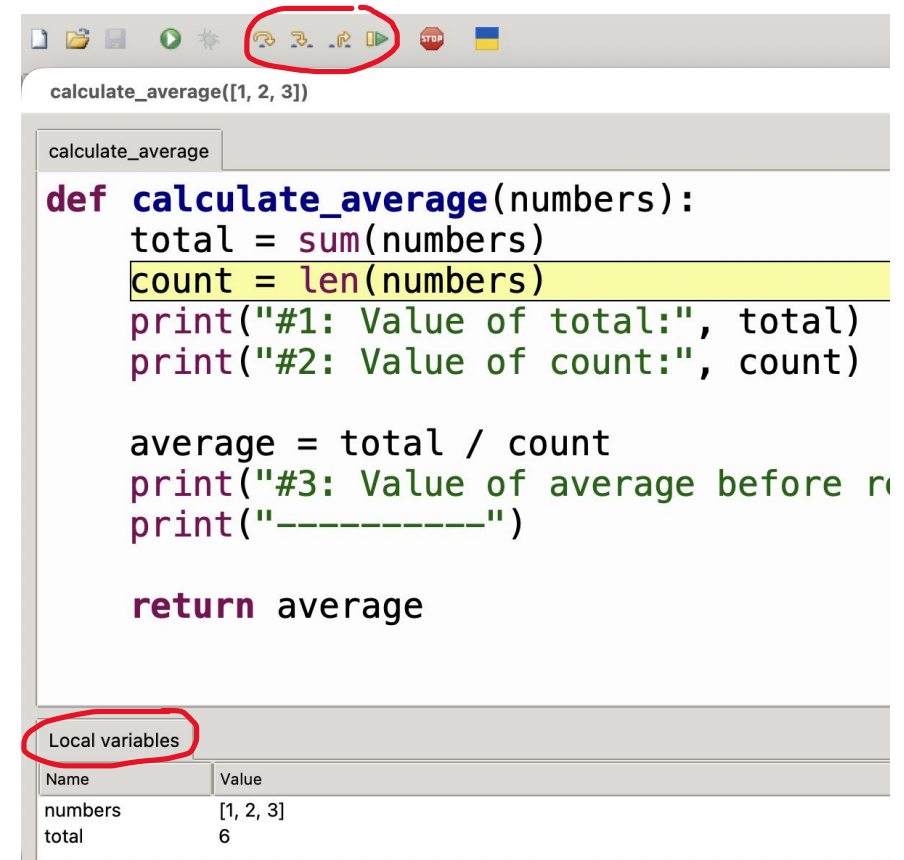
Sidebar: Debuggers



Thonny's debugger lets you execute code one expression or statement at a time (not just line-by-line), and shows the live variable state (global and local) as it goes.

It highlights the current expression before it evaluates and shows the result after. You can also choose how to 'step' through the code.

If you would like to learn more about how to use Thonny's debugger, check out the bonus slides on the course website.



Logical Errors

Logical Error Example

The following code has a logical error. Can you find it?

```
1 """
2 Counts the number of passing grades in a list, where a passing grade is 60 or above.
3 Parameters: grades (list of int/float): A list of grade values.
4 Returns: int: The count of grades that are 60 or higher.
5 """
6
7 def count_passing(grades):
8     count = 0
9     for i in range(len(grades) - 1):
10         if grades[i] >= 60:
11             count = count + 1
12     return count
13
14 print(count_passing([55, 70, 80, 90]))
```

Let's run it in Thonny!

Logical errors won't give you an error; the program will run normally, but produce the wrong result.

How do we catch a logical error? We have to build a trap ourselves! We use `asserts` to force an error when our assumptions about the program's behavior turn out to be wrong.

Once we know a logical error exists, we can use `print` statements to find why and where it's happening, then decide how to fix it.

Let's demonstrate with the code from the previous slide.

Summary: The debugging approach

1. Get it to parse. Fix every syntax error first, nothing else can be tested until Python can parse the file.

 **2. Get it to run.** Fix runtime errors next, get the program to complete without crashing.

 **3. Get it right.** Only now hunt for logical errors, comparing actual output to expected output and using `print` statements to trace your program execution.

Let's try our debugging skills on a function

```
"""
Calculates the average of all temperatures in the list that are considered
"hot" (greater than or equal to 90 degrees).

Parameters: temps (list of int/float): A list of temperature readings.

Returns: float: The average of all temperatures above 90 degrees.
"""
```

```
def average_hot_temp(temps:
    count = 0
    total = 0
    for i in range(len(temps)):
        if temps[i] > 90:
            count = count + 1
            total = total + temps[i + 1]
    return total / count

average_hot_temp([72, 95, 90, 88, 99])
```

Debugging Project

Let's debug a bigger project.

Let's apply our debugging approach to the following buggy project:

```
"""
```

```
Shopping Cart Mini-Project (BUGGY VERSION)
```

```
-----
```

```
Calculates a cart subtotal.
```

```
Each item is a list: [name, price, quantity]
```

```
The cart is a list of items: [item1, item2, item3, ...]
```

```
"""
```

Learning Goals

- Interpret **error messages** generated by Python programs.
- Use **debugging** on a Python program that is not working correctly to identify where the problem occurs and fix simple bugs.