#2-2: Errors, Debugging, and Testing

CS SCHOLARS – PROGRAMMING

Learning Goals

Recognize the different types of **errors** that can be raised when you run Python code

Debug logical errors by using **debugging strategies**

Write **tests** that verify whether a program is working as expected

Python Errors

Syntax Errors Occur due to Bad Syntax

When Python executes your code, it first has to break your text down into **tokens**, then **structure** those tokens into a format that the computer can execute.

The programming language's **syntax** is a set of rules for how code instructions should be written. When syntax is correct, Python is able to tokenize and structure code without a problem.

If the interpreter runs into an error while tokenizing or structuring, it calls that a **syntax error**. In other words, you get a syntax error when the code you provide does not follow the rules of the Python language's syntax.

A syntax error means that **none of your code will run**, because the syntax can't be parsed.

Examples of Syntax Errors

Most syntax errors are called **SyntaxError**, which make them easy to spot. For example:

```
x = @      # @ is not a valid token
4 + 5 = x # the parser stops because it doesn't follow the rules
```

There are two special types of syntax errors: IndentationError and incomplete error.

```
    x = 4    # IndentationError: whitespace has meaning
print(4 + 5 # Incomplete Error: always close parentheses/quotes
```

Execution Errors are Runtime Errors

After Python tokenizes and structures the code, the interpreter runs through the control flow of the program line-by-line.

If an error occurs as the code is being executed, it's called a **runtime error**. Everything that happened before that error will execute just fine, but everything afterwards will not run.

Runtime errors have many different names in Python. Each name says something about what kind of error occurred, so reading the name and text can give you additional information about what went wrong.

Examples of Runtime Errors

```
print(Hello) # NameError: used a missing variable

print("2" + 3) # TypeError: illegal operation on types

x = 5 / 0 # ZeroDivisionError: can't divide by zero
```

We'll see more types of runtime errors as we learn more Python syntax.

Other Errors are Logical Errors

If we manage to run Python code completely, does that mean it's correct?

Not necessarily! **Logical errors** can occur if code runs but produces a result that was not what the user intended. The computer can't catch logical errors because the computer doesn't know what we intend to do.

To catch logical errors, you usually need to **test** your code. We'll do this mainly with `assert` statements.

Examples of Logical Errors

```
print("2 + 2 = ", 5) # no error message, but wrong!


def double(x):
    return x + 2 # adding instead of multiplying

assert(double(3) == 6) # 6 is the intended result
```

`assert` Statements Check Correctness

An `assert` statement takes a Boolean expression. If the expression evaluates to `True`, the statement does nothing. If it evaluates to `False`, the program crashes.

We use `assert` statements to check for logical errors by testing whether the output of a function call is equal to what we expect it to be. If the result is not correct, you get an `AssertionError`.

```
assert(findAverage(20, 4) == 5)
```

Activity: Predict the Error Type

Let's test your knowledge of error types with another Kahoot!

Given a line of code, predict whether it will result in a Syntax Error, Runtime Error, Logical Error, or no error.

If you aren't sure, try to think about whether the problem will occur during syntax parsing/structuring, or execution, or if it will run properly but still have a problem.

Join at kahoot.it

Debugging

Error Messages Can Help... Sometimes

In the second lecture we talked about how to read **error messages** to debug syntax and runtime errors:

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 gives you information about what went wrong.

example.py

```
1 print(Hello World)
2 Print("Hello World")
```

```
Running script: "C:\Users\river\Downloads\example.py"
  File "C:\Users\river\Downloads\example.py", line 1
    print(Hello World)
                  ^
SyntaxError: invalid syntax

>>>
```

inline arrow

line number

example.py

```
1 print("Hello World")
2 Print("Hello World")
```

```
Running script: "C:\Users\river\Downloads\example.py"
Hello World
Traceback (most recent call last):
  File "C:\Users\river\Downloads\example.py", line 2, in
<module>
    Print("Hello World")
NameError: name 'Print' is not defined

>>>
```

error type

Debug Logical Errors By Checking Inputs and Outputs

When your code generates a logical error, the best thing to do is **compare the expected output to the actual output**.

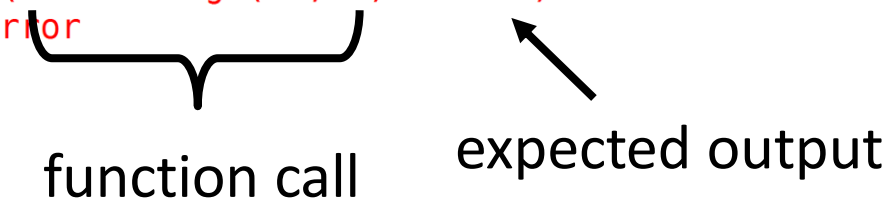
1. Copy the function call from the `assert` that is failing into the interpreter. Compare the actual output to the expected output.
2. If the **expected** output seems incorrect, re-read the problem prompt.
3. If you're not sure why the actual output is produced, use a **debugging process** to investigate.

```
def findAverage(total, n):
    if n <= 0:
        return "Cannot compute the average"
    return total // n

assert(findAverage(13, 2) == 6.5)
```

```
Running script: "C:\Users\river\Desktop\example.py"
Traceback (most recent call last):
  File "C:\Users\river\Desktop\example.py", line 6, in
<module>
    assert(findAverage(13, 2) == 6.5)
AssertionError

>>>
```



Understanding the Prompt

When something goes wrong with your code, before rushing to change the code itself, you should make sure you understand **conceptually** what your code does.

First make sure you're solving the right problem! Re-read the problem prompt to check that you're doing the right task.

It can often help to analyze the **test cases** to make sure you understand why each input results in each output. We'll talk more about these a bit later in the lecture.

Ways to Debug

There are many approaches you can take towards debugging code effectively. Let's highlight three.

Rubber Duck Debugging: talking through your code

Printing and Experimenting: visualizing what's in your code

Thorough Tracing: checking each part of the code line-by-line

Rubber Duck Debugging

If you find yourself getting stuck, try **rubber duck debugging**. Explain what your code is supposed to do and what is going wrong out loud to an inanimate object, like a rubber duck.

In the process of explaining your code out loud to someone else you may find that a piece of your code does not match your intentions, or that you missed a step. You can then make the fix easily. This works more often than you might think!

Print and Experiment

If rubber duck debugging doesn't work, try **printing and experimenting** to determine where in your code the problem is.

Add print statements around where you think the error occurs that display relevant values in the code. Run the code again and check whether the printed values match what you think they should be at that stage in the code.

Each print call should also include a brief string that gives context to what is being printed. For example:

```
print("Result pre-if:", result)
```

Making Hypotheses

If something looks wrong in the printed results, make a hypothesis about what the problem is and adjust your code accordingly. Then run the code again and see if the values change. Repeat this as much as necessary until your code works as expected.

An important part of this process is that you have to be intentional about the changes you make. Don't just change parts of the code haphazardly - have a theory for why each change might fix your problem.

Activity: Debug getSize

These functions are supposed to calculate the largest root of a quadratic function. As a reminder, given a function **$ax^2 + bx + c$**, we can find its two roots with:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Our program has some bugs. Work with a group to debug the program. Try using either rubber duck debugging or print and experiment to figure out what's going wrong.

```
import math
def helper(a, b, c):
    return math.sqrt(b*2 - 4*a*c)

def getBiggestRoot(a, b, c):
    root1 = (-b + helper(a, b, c)) / 2 * a
    root2 = (-b - helper(a, b, c)) / 2 * a
    if root1 > root2:
        return root1
    else:
        return root2

assert(getBiggestRoot(1, -1, -12) == 4)
# roots: 4 and -3
assert(getBiggestRoot(6, -2, -20) == 2)
# roots: 2 and -5/3
```

Thorough Tracing

If you can't find the problem through printing and experimenting, you may have to resort to **thorough tracing** to determine what's going wrong.

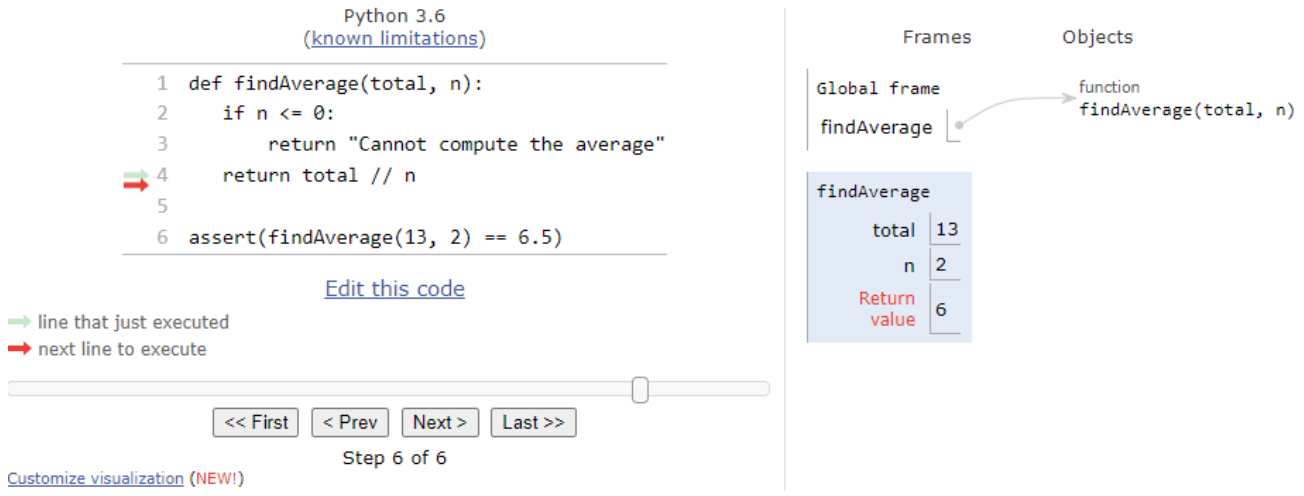
Step through your code line by line and track on paper what values should be held in each of your variables at each step of the process.

Compare your traced values with what you would create step-by-step if you were solving the problem by hand. This might help you identify where the problem is occurring.

Tracing with Tools

Learning how to trace code by hand is a useful skill, but there are also **tools** that can help support you during debugging. Start with the website http://pythontutor.com/ .

If you paste your code into the editor and click 'Visualize Execution', you can step through your code line by line. The tool will visualize the **state** of the program on the right as you step through it. This can be very helpful!



Activity: Practice with PythonTutor

Here is a function that is supposed to take a shirt size in inches and return the size as a string (small, medium, or large). But it's not working correctly.

Try pasting the code into PythonTutor and stepping through the program line by line.

Link: http://pythontutor.com/

What do you notice as you're tracing the program? What stands out?

```
def getSize(length):
    if length <= 38:
        print("small")
    elif length <= 40:
        print("medium")
    else:
        print("large")
    return length


assert(getSize(39) == "medium")
```

Debugging is Hard

Finally, remember that debugging is hard! If you've spent more than 15 minutes stuck on an error, more effort is not the solution. Get a friend to help, or take a break and come back to the problem later. A fresh mindset will make finding your bug much easier.

Testing

Test Cases Use `assert` Statements

In this week's homework (and future homeworks), the starter file for programming assignments will contain test cases that use `assert`.

To check your solutions against the test cases, use **Run File as Script**. If you have not commented out the test cases, the file runs without crashing, and all the outputs look correct, your code is (probably) correct. If your code throws an `AssertionError`, that means you have a **logical error** in one or more of your solutions.

Writing Your Own Tests

In real life (unlike homework assignments), test cases aren't provided for you. You have to write your own tests if you want to make sure that your code works properly.

In general, you want to have a set of tests for **every function** that you write. Designing those tests is a bit of an art form!

Testing

When writing test functions, you need to cover **likely cases where things can go wrong**. If you don't, your program might develop a bug without you realizing!

In particular, you should always try to cover:

- **Normal cases** – provided and obvious examples
- **Large cases** – larger-than-usual input
- **Edge cases** – pairs of input that result in opposite choices in the code
- **Special cases** – 0 and 1, empty string, unexpected types
- **Varying results** – make sure that all your test cases don't return the same result!

Example Test Cases

Recall the car rental program we wrote last time. Let's write some test cases for it!

```
assert(canRentCar(23, True) == False) # normal case
assert(canRentCar(87, False) == False) # large input
assert(canRentCar(25, True) == False) # edge case
assert(canRentCar(26, True) == True) # edge case, varying result
assert(canRentCar(-32, True) == False) # special case
assert(canRentCar(32, True) == True) # varying result
```

Testing `gradeCalculator`

You do: Try to come up with test cases for each of these categories for a function `gradeCalculator`, which turns numeric grades into letter grades, returning the letter grade as a string.

Normal case:

Large case:

Edge case:

Special case:

Varying results:

Test first!

There's a temptation when programming to write the code first, then test it when you're done.

It's actually much more useful to write the tests first, then write the code! Writing the tests will help you better understand what the code needs to do.

This is called **test-driven development**.

Learning Goals

Debug logical errors by using **debugging strategies**

Write **tests** that verify whether a program is working as expected

Apply general **style principles** to write **clear** and **robust** code