#4-1: Debugging, Testing, and Style

CS SCHOLARS – PROGRAMMING

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

Python Errors

Previously, we talked about the three types of errors you encounter while coding: **syntax**, **runtime**, and **logical**.

It's not enough to recognize what these errors are – as a programmer, you have to have a strategy for how to **fix** these errors when you encounter them as well!

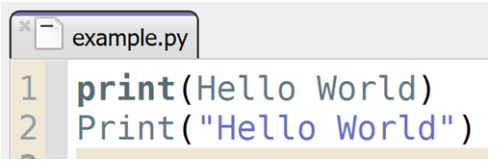
Better yet, it's good to use coding approaches that reduce the number of errors you encounter, like **testing** and **style**.

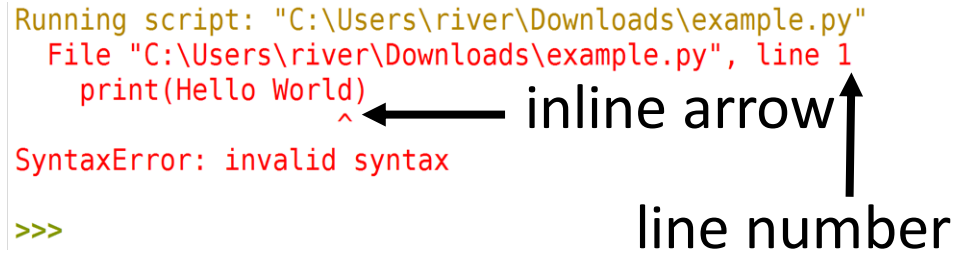
Debugging

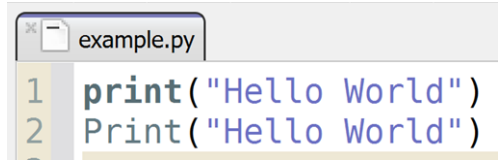
Error Messages Can Help... Sometimes

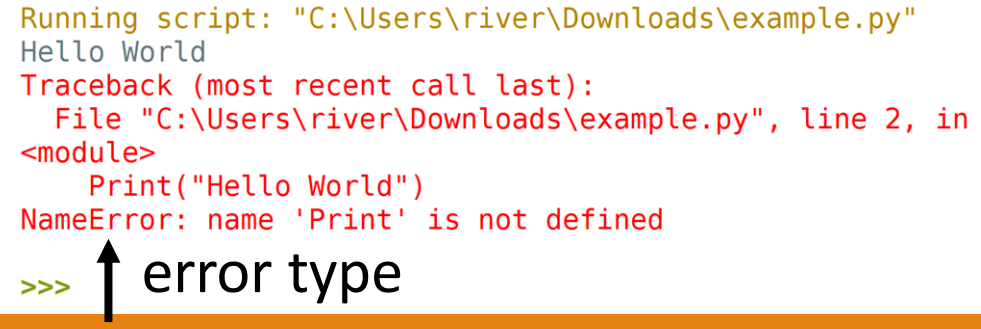
At the very beginning of the program, we talked about how to read **error messages** to debug syntax and runtime errors:

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.









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**.

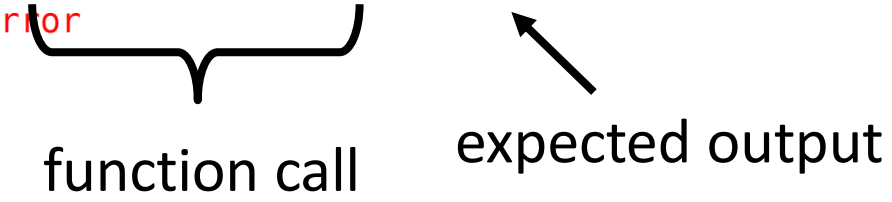
1. Copy the function call from the `assert` that is failing into the interpreter. Compare the actual output to the expected output.
2. If the **expected** output seems incorrect, re-read the problem prompt.
3. If you're not sure why the actual output is produced, use a **debugging process** to investigate.

```
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

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.

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.

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

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

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.

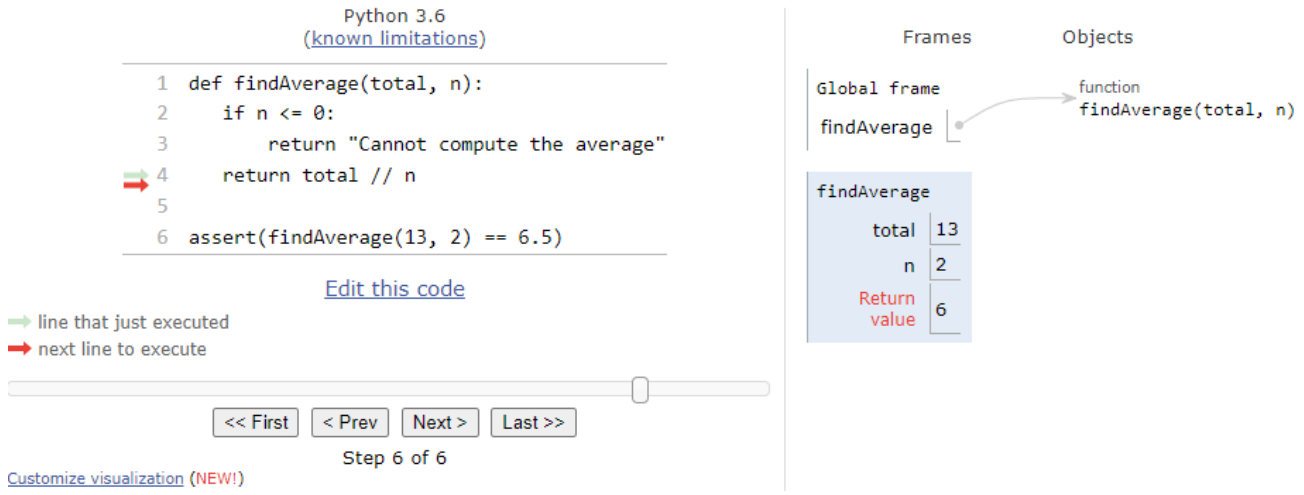
Step through your code line by line and track on paper what values should be held in each of your variables at each step of the process.

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

You do: Here is a new buggy program. This one is supposed to take two lists, multiply each pair of elements at the same index, and return a list holding the results, but it has a bug.

Try pasting the program 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 multiplyLists(lst1, lst2):
    result = []
    for i in range(len(lst1)):
        for i in range(len(lst2)):
            result.append(lst1[i] * lst2[i])
    return result

assert(multiplyLists([1, 2, 3],
                     [4, 5, 6]) == \
       [4, 10, 18])
```

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

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 `isPrime` program we wrote earlier in the program. Let's write some test cases for it!

```
assert(isPrime(18) == False) # normal case
assert(isPrime(37) == True) # varying result
assert(isPrime(29*37) == False) # large input
assert(isPrime(2) == True) # edge case
assert(isPrime(25) == False) # edge case
assert(isPrime(1) == False) # special case
```

Testing `findMax`

You do: Try to come up with test cases for each of these categories for a function `findMax`, which takes a list of numbers and returns the largest number in the list.

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**.

Style

Real-World Coding

When you're working on a homework assignment, you probably mainly care about getting the code to work.

But this isn't how programming works in real life. If you write a piece of code that accomplishes a task, it's highly likely that you or someone else will want to use that code again at some point in the future.

It's even possible that you'll want to change the code slightly when the goals of the task change.

Purpose of Style

Whenever you write code that anyone (including yourself) will look at again in the future, you should write that code with good **style**.

Style is all the decisions you make as you write code about how to organize and implement an algorithm.

It's very much like a writing style or a drawing style; everyone will approach how they organize their code a little differently.

Different Styles

```
# Input: int
# Output: bool
def is_prime(num):
    if num < 2:
        return False
    for factor in range(2, num):
        if num % factor == 0:
            return False
    return True
```

```
def isPrime(x):
  """
  takes an integer and returns
  whether or not it's prime
  """
  if(x<=1):
    return(False)
  # check each possible factor
  for i in range(2,x):
    if((x%i)==0):
      return(False)
  return(True)
```

Style Principles

There are lots of recommendations for how to write code with good style. We'll group them into two major categories:

- **Clarity** – principles that make your code easier to read
- **Robustness** – principles that make code easier to modify

Discuss: what could go wrong if your code is not clear? What if it's not robust?

Style – Clarity

Style Principles for Clarity

You spend as much time reading code as you do writing code, if not more! Writing code that is **clear** and easy to read is therefore extremely important.

We'll look at four general principles for writing clear code:

1. Use consistent formatting
2. Use good naming conventions
3. Don't include unnecessary code
4. Remember to document

Consistent Formatting

In general, try to be **consistent** with how you format whitespace in code.

Python will let you get away with somewhat uneven indentation in different parts of a program, but the result is harder to read. Be consistent about whether you use spaces or tabs, and always use the same number of spaces or tabs when indenting code.

BAD

```
def isPrime(x):
    if x < 2:
      return False
    for factor in range(2, x):
     if x % factor == 0:
              return True
    return False
```

GOOD

```
def isPrime(x):
    if x < 2:
        return False
    for factor in range(2, x):
        if x % factor == 0:
            return True
    return False
```

Consistent Formatting

Code is also easier to read when the whitespace used between tokens is consistent.

You can choose to use no unnecessary whitespace, or add a space between every pair of tokens, or even choose some operators that will have whitespace added and some that won't.

BAD

```
x =(3+  2 )   / 5
```

GOOD

```
x = (3 + 2) / 5
```

Also- don't let your lines of code get too long. A general guideline is to pick a number of characters - let's say 80 - and make sure every line of code you write is no longer than that length. Thonny lets you place an indicator in the editor at that location.

Good Naming Conventions

It's important to give your variables descriptive names that describe the data held by the variable. Having descriptive, meaningful names will make understanding code much easier.

BAD

```
def isPrime(a):
    if a < 2:
        return False
    for b in range(2, a):
        if a % b == 0:
            return True
    return False
```

GOOD

```
def isPrime(num):
    if num < 2:
        return False
    for factor in range(2, num):
        if num % factor == 0:
            return True
    return False
```

There are a few cases where seemingly-meaningless variable names have gained meaning over time, usually when they are shorthand for a longer word. For example, we often use x, y, and n for numbers. These are okay to use when there's no greater meaning behind the variable.

Avoid Unnecessary Code

Unnecessary code is code that will never actually be run by Python, or code that Python runs but never uses in a meaningful way (like a variable that is defined, then never used). We also refer to this as dead code.

BAD

```
def isPrime(num):
    if num < 2:
        return False
    end = num
    for factor in range(2, num):
        if num % factor == 0:
            return True
        else:
            pass
    return False
    return
```

GOOD

```
def isPrime(num):
    if num < 2:
        return False
    for factor in range(2, num):
        if num % factor == 0:
            return True
    return False
```

Unnecessary code won't actually harm your program, but it does make the program more complicated to understand.

Document Your Code

Finally, make sure to document your code using comments! We haven't talked much about *when* to write comments. In general, comments are most useful when they explain something that is not immediately obvious from the code itself.

Consider this code snippet, from isPrime. This is a good comment because it clarifies something that might not be immediately obvious- we intentionally skipped num because it's okay for a prime number to divide itself.

GOOD

```
def isPrime(num):
    if num < 2:
        return False
    # do not iterate over 1 or num because prime
    # numbers are divisible by themselves and 1
    for factor in range(2, num):
        if num % factor == 0:
            return True
    return False
```

Activity: Find Style Errors

You Do: What are some **clarity** style errors in this piece of code?

```
def sumToN(n):
    tmp = 0
    for abc in range(n):
        tmp += abc
        abc=abc+1
    return tmp
```

Style – Robustness

Style Principles for Robustness

There are also several principles that will help you write **robust** code. This will make your code easier to change and update over time, and decrease the chance of bugs occurring.

We'll look at four general principles for writing robust code:

1. Avoid repetitive code
2. Avoid magic numbers
3. Join up related conditionals
4. Test all functions

Avoid Repetitive Code

First, try not to write **repetitive** code. This is code where similar logic is repeated over many lines instead of being condensed into a single structure.

When you find yourself repeating code- and **especially** when you find yourself copying and pasting code – look for the pattern and move it into a loop or a generalized action. Helper functions can be useful here too.

BAD

```
def coordToRow(x):
    if x < 50:
        return 0
    elif x < 100:
        return 1
    elif x < 150:
        return 2
    elif x < 200:
        return 3
```

GOOD

```
def coordToRow(x):
    for row in range(4):
        if x < (row+1) * 50:
            return row
```

ALSO GOOD

```
def coordToRow(x):
    return x // 50
```

Avoid Magic Numbers

Second, avoid using **magic numbers**. These are numbers used somewhere in an algorithm for no clear reason, without being stored in a variable first.

Magic numbers are mainly a problem when it comes to updating code. Consider what would be required to change the size of the grid cells in these two implementations.

BAD

```
def drawGrid(canvas, canvasSize):
    for row in range(4):
        top = row * 50
        bottom = top + 50
        for col in range(4):
            left = col * 50
            right = left + 50
            canvas.create_rectangle(left, top,
                                    right, bottom)
```

GOOD

```
def drawGrid(canvas, canvasSize):
    rows = 4
    cellSize = canvasSize / rows
    for row in range(rows):
        top = row * cellSize
        bottom = top + cellSize
        for col in range(rows):
            left = col * cellSize
            right = left + cellSize
            canvas.create_rectangle(left, top,
                                    right, bottom)
```

Non-Magic Numbers

Not every number is a magic number. For example, to get the ones digit of a number you have to mod by 10. In this case, it's pretty clear why 10 is used, and you're not likely to change it to anything else in the future, so you don't need to store 10 in a variable.

```
def getOnesDigit(num):
    return num % 10
```

0, 1, 2, and 10 are often (though not always) safe to use directly in code.

Join Up Conditionals

Third, make sure to **join up conditionals** as appropriate. If you have multiple conditional checks that are happening in a row and only one of them should be visited, those checks should form one `if-elif-else` block, not several independent `if`s.

BAD

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

GOOD

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

The main problem with not joining up related conditionals is that you might accidentally enter more than one conditional branch if you aren't careful with the tests. It's just safer to combine them all together.

Test Your Functions

Finally, make sure to **write test functions** for each function you implement! Yes, writing test cases takes time and can be tedious, but it will help you out a lot in the long run.

Test functions are primarily useful at two points in time. The first is naturally when you first write a function. The test function ensures that it's working the way you want it to.

But test functions are also useful later on, if you need to modify a function. Having an active test suite makes it easy to check whether a new modification breaks any of the previous requirements of the program.

Activity: Find Style Errors

You Do: What are some **robustness** style errors in this piece of code?

```
def getSize(length):
    size = "small"
    if 38 < length and length <= 40:
        size = "medium"
    if 40 < length:
        size = "large"
    return size
```

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