Testing and Debugging

15-110 – Monday 02/03

Learning Goals

- Write **test cases** to determine whether code works properly
- Debug syntax and runtime errors by **interpreting error messages**
- Debug runtime and logical errors by using the **scientific method**

Testing and Debugging Are As Important as Coding

We spend a lot of time on how to design algorithms and write code, but that's only two parts of what programmers do.

It's equally important that you **make sure your program works correctly**. And when it doesn't, you have to **figure out how to fix it**.

You'll need to use testing & debugging in this class on your own code, but also in real life if you interact with an already-written program that doesn't work quite correctly.

Testing

Test Cases Check Specific Scenarios

You could test your code by checking whether it outputs the correct result on every possible input. But that's too much work!

Instead, we design a set of **test cases,** where each test case checks if the output is correct on a specific input. Inputs are chosen to cover different possible scenarios.

If we select a set of test cases that have broad coverage, we can be fairly sure that our program works.

Test Output with `assert` Statements

To write test cases, we'll use **`assert` statements**. An `assert` takes a Boolean expression as input, does nothing if the expression evaluates to `True`, and raises an `AssertionError` if the expression evaluates to `False`.

```
assert(4 > 2) # does nothing
assert(3 == 5) # raises a runtime error
```

Input, Actual Output, and Expected Output

To write a test case, you need to represent the three core parts of a test – the **input**, the **actual output**, and the **expected output**.

$$\texttt{assert(}\underbrace{\texttt{sum1toN(}\overbrace{\texttt{10}}^{\text{input}}\texttt{)}}_{\text{actual output}}\texttt{ == }\overbrace{\texttt{55}}^{\text{expected output}}\texttt{)}$$

Test Cases Should Cover Different Scenarios

A proper set of test cases should cover all the different types of inputs the program might encounter.

Normal Case: A typical input that should follow the main path through the code.

Edge Case: A pair of inputs that test different choice points in the code. So if a condition in the problem checks whether `n < 2`, two important inputs are 1 and 2.

Special Case: A 'default' input that may behave differently from normal cases. In this class, we've seen the default values `0`, `1`, and `""`.

Varying Results: Test cases should cover multiple possible results. This is especially important for Boolean functions, which should check results of `True` and `False`.

Large Input Case: A typical input, but of a larger size than usual.

Example: Test Cases for `digitCount(num)`

Let's make a test case set for a function `digitCount(num)`, which counts the number of digits in an integer.

Normal Case: `assert(digitCount(1234) == 4)`
Edge Case: `assert(digitCount(7) == 1)`
Special Case: `assert(digitCount(0) == 1)`
Varying Result: `assert(digitCount(20) == 2)`
Large Input Case: `assert(digitCount(54365463734365) == 14)`

You may need several tests of each type to get broad coverage of all possible scenarios.

Activity: Design Tests for `isPrime(num)`

You do: What tests should we write for a function `isPrime(num)`, which returns `True` if the given integer is prime and `False` otherwise?

Interpreting Errors

Syntax, Runtime, and Logical Errors

In the first week, we discussed the three types of errors Python can encounter:

Syntax Errors, which happen when Python can't parse code

Runtime Errors, which happen when the interpreter crashes while running code

Logical Errors, which happen when code doesn't work correctly

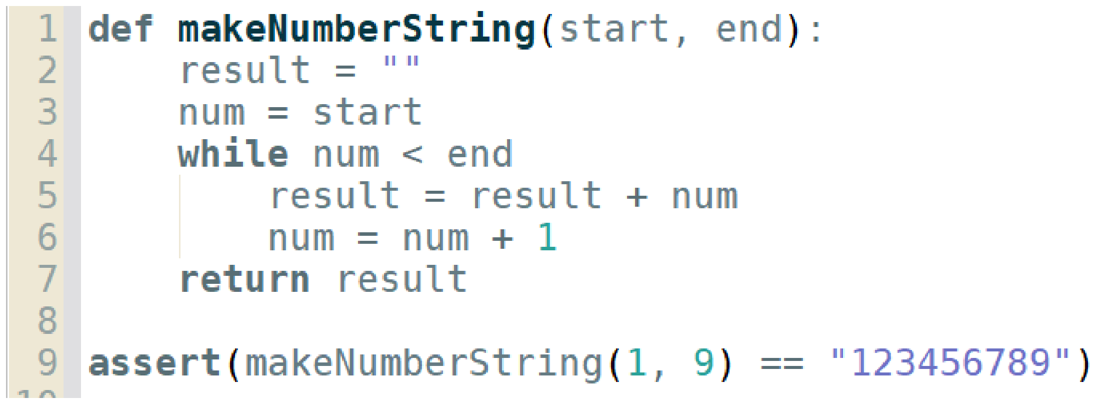
We'll use slightly different approaches to debug these three types of errors.

Debug Syntax Errors By Reading the Message

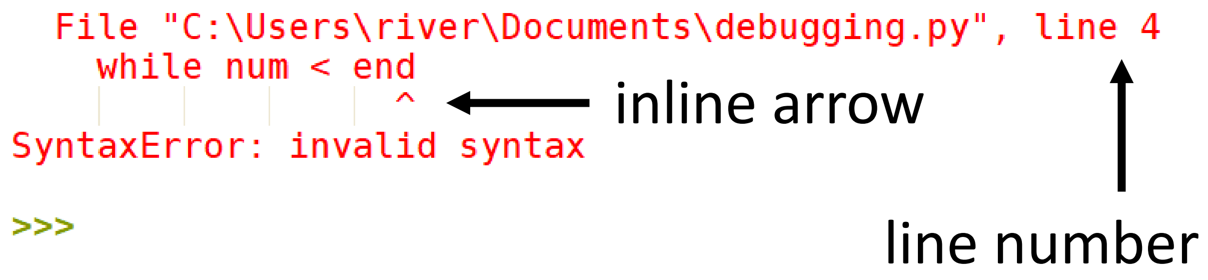
When your code generates a `SyntaxError`, the best thing to do is **read the error message**.

1. Look for the **line number**. This line tells you approximately where the error occurred.
2. Then look for the **inline arrow**. The position gives you more information about the location.
3. If you're not sure why a syntax error would occur there, **compare your code** to example code from the course slides.

The location Python suggests isn't always correct – sometimes the error happens in the lines **before** the suggested location instead.



```
1  def makeNumberString(start, end):
2      result = ""
3      num = start
4      while num < end
5          result = result + num
6          num = num + 1
7      return result
8
9  assert(makeNumberString(1, 9) == "123456789")
```



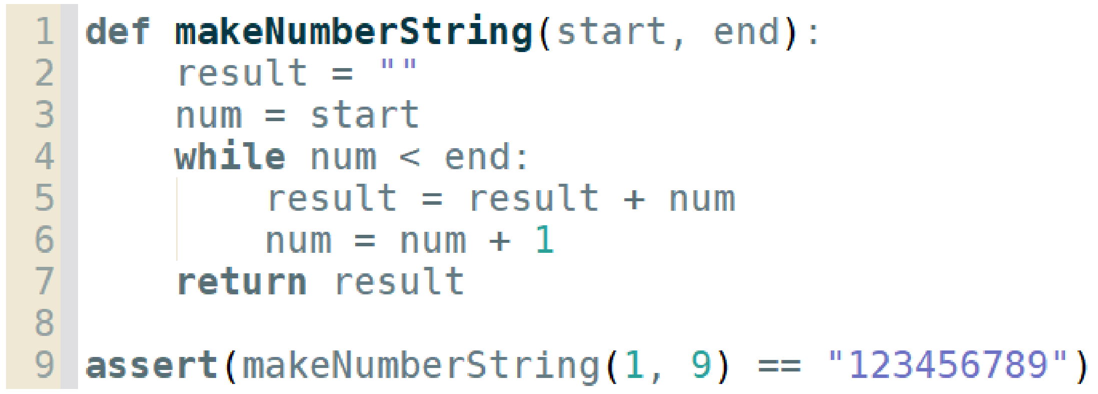
```
  File "C:\Users\river\Documents\debugging.py", line 4
    while num < end
                  ^
SyntaxError: invalid syntax

>>>
```

Debug Runtime Errors By Reading the Message

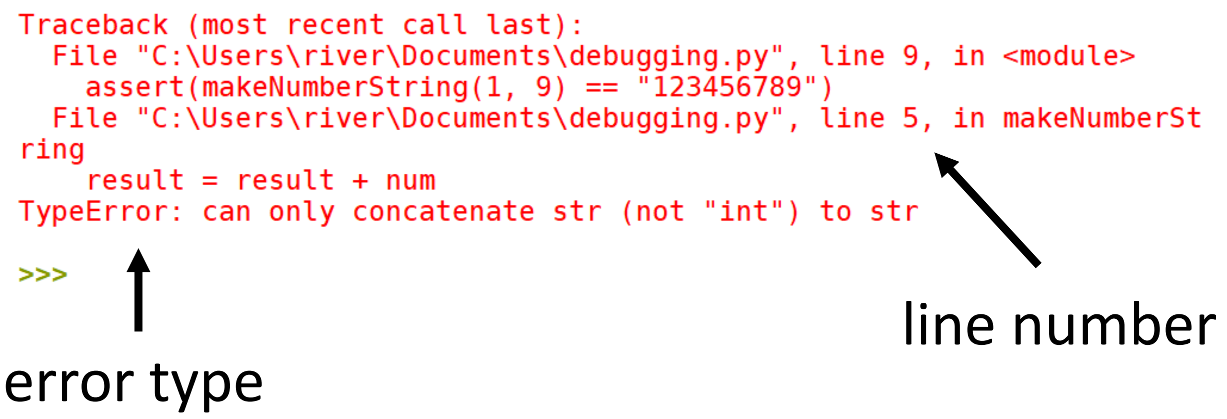
When your code generates a runtime error, the best thing to do is **read the error message**.

1. Look for the **line number**. This line tells you approximately where the error occurred.
2. Then look for the **error type**. The error type and its message gives you information about what went wrong.
3. If you're not sure why that error would occur on that line, use the **debugging process** to investigate.

Any error that is not a `SyntaxError`, `IndentationError`, or `AssertionError` is a runtime error.



```
1  def makeNumberString(start, end):
2      result = ""
3      num = start
4      while num < end:
5          result = result + num
6          num = num + 1
7      return result
8
9  assert(makeNumberString(1, 9) == "123456789")
```



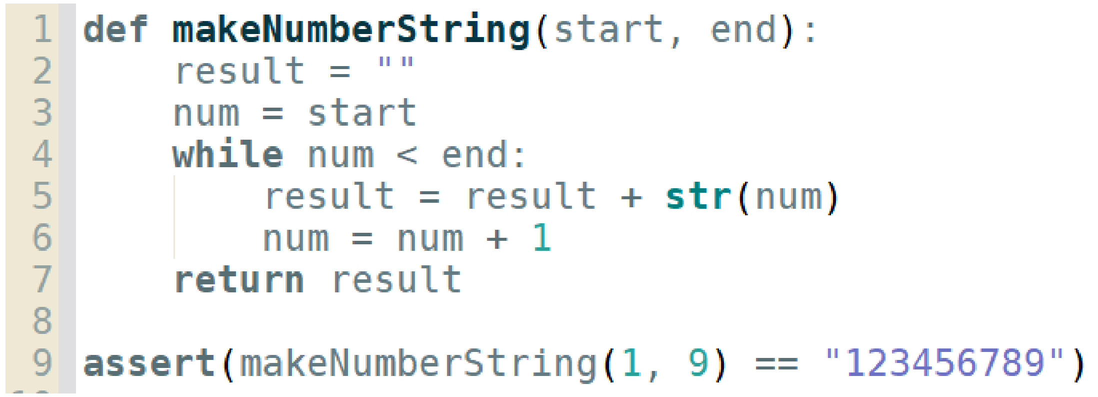
```
Traceback (most recent call last):
  File "C:\Users\river\Documents\debugging.py", line 9, in <module>
    assert(makeNumberString(1, 9) == "123456789")
  File "C:\Users\river\Documents\debugging.py", line 5, in makeNumberSt
ring
    result = result + num
TypeError: can only concatenate str (not "int") to str

>>>
```

Debug Logical Errors By Checking Inputs and Outputs

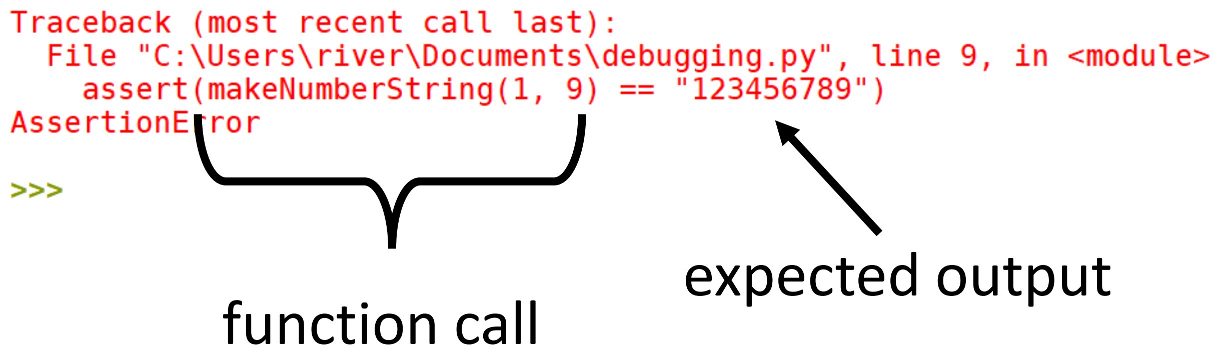
When your code generates a logical error, the best thing to do is **compare the expected output to the actual output**.

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 the **debugging process** to investigate.

If you've written the test set yourself, you should also take a moment to make sure the test itself is not incorrect.



```python
def makeNumberString(start, end):
    result = ""
    num = start
    while num < end:
        result = result + str(num)
        num = num + 1
    return result

assert(makeNumberString(1, 9) == "123456789")
```



```
Traceback (most recent call last):
  File "C:\Users\river\Documents\debugging.py", line 9, in <module>
    assert(makeNumberString(1, 9) == "123456789")
AssertionError

>>>
```

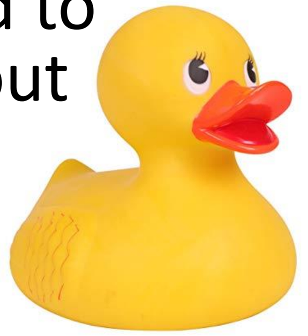
Debugging Process

Understanding Your Code

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.

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. Sometimes, saying things out loud will help you realize what's wrong.



Debug with the Scientific Method

When you're trying to debug a tricky error, you should use a process similar to the **scientific method**. We'll reduce it down to five core steps:

1. Collect data
2. Make a hypothesis
3. Run an experiment
4. Observe the result
5. Repeat the process (if necessary)

Step 1: Collect Data

First, you need to **collect data** about what your code is currently doing.

You can already see the steps of your algorithm, but you can't see how the variables change their values while the program runs. Add **print statements** at important junctures in the code to see what values the variables hold.

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

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

Step 2 & 3: Make a Hypothesis; Experiment

At a certain point, you should see something in the values you are printing that is unexpected. At that point, **make a hypothesis** about why the variable is holding that value.

Once you have a hypothesis, **test it** by making an appropriate change in your code. For example, if you think the code never enters an if statement, add a print to the beginning of the conditional body to see if it gets printed.

Note: do not change things randomly, even if you get frustrated! Even if it makes you code work on one test, it might start failing another.

Step 4: Observe the Result

Once you've made the change, **observe the result** by checking the new output of your code.

Print statements are still helpful here. You can also use **variable tables** to see how a variable's behavior changes before vs. after the experiment, by writing out the value in a variable at each juncture of the code by hand.

For particularly tricky code, there are **online visualization tools** that let you see how your code behaves step-by-step. Here's one we recommend: pythontutor.com/visualize.html

Step 5: Repeat As Necessary

Finally, know that you may have to **repeat** the debugging process several times before you get the code to work.

This is normal; sometimes bugs are particularly hard to unravel, and sometimes there are multiple different bugs between your code and a correct solution.

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 or TA to help, or take a break and come back to the problem later. A fresh mindset will make finding your bug much easier.

Learning Goals

- Write **test cases** to determine whether code works properly
- Debug syntax and runtime errors by **interpreting error messages**
- Debug runtime and logical errors by using the **scientific method**