#1: Data and Debugging

SAMS SENIOR NON-CS TRACK

Course Logistics

Course Goals and Expectations

Course Website: http://krivers.net/SAMS-m19/

Staff & Student Introductions

Course Plan

Schedule here: http://www.krivers.net/SAMS-m19/schedule_seniorB.html

Tuesday and Thursday lectures will combine lectures & in-class exercises

Week 1: Basics

Week 2: Graphics

Week 3: Interaction

Week 4: Conditions

Week 5: Iteration

Week 6: Creativity

Succeeding in this Course

You will be evaluated based on class participation and in-class exercises.

Participation means attending class each day and staying involved in class discussion/activities.

In-class exercises are assigned in class each day, and due at the end of class. You may collaborate on exercises as much as you want, but must write your own code yourself- no copying! Exercise code should be submitted on Autolab, and will be graded based on effort & accuracy.

Want help reviewing the material? Come to office hours! They occur Monday-Thursday from 12-1pm (GHC 4109) and from 6:30-8:30pm (Gates 5th floor teaching commons).

Things You'll Need

Python 3: https://www.python.org/

Pyzo: http://www.pyzo.org/ (or another IDE of your choice)

Both can be found on CMU cluster computers if you don't have a laptop.

Today's Learning Goals

Understand what a programming language is

Write code that uses numbers, text, and operations to compute simple expressions

Determine when a program isn't working, identify the bug, and fix it

Programming

Programming: How We Talk to Computers

We know how to give instructions to other human beings- we just tell them what they need to do, step by step. We do this all the time with recipes.

However, writing a recipe can be difficult. Can you assume someone knows how to grease a pan? What does it mean to salt something 'to taste'?

Writing a program for a computer is like writing a recipe for someone who is new to cooking. Each step needs to be specified precisely, with no ambiguity.

This means we can't communicate with a computer using natural language, as natural language is full of ambiguities! Instead, we use a specially-created language that the computer understands.

Calculator Instructions

What happens when I type this into a calculator?

```
4 + 16 / 2
```

The calculator knows rules for how to **parse** and **evaluate** mathematical expressions

A programming language like Python is like a calculator, except that it can handle **much more complex instructions**

Editor vs. Interpreter

A **program editor** is just a text editor that lets you write programs, save them to files, and run them.

An **interpreter** is the place where the program is actually **parsed** and **evaluated**. In Python, we can choose to write code directly in the interpreter.

We generally use the interpreter when experimenting and the editor when writing code we want to save or run multiple times.

Let's try out our main editor, Pyzo...

Annotating Code

As we start writing code, we might want a way to add notes to our code that explain what we're doing.

You can add **comments** to Python code by starting a line with the symbol **#**. Python will ignore anything that follows this character on the same line.

Examples:

```
# This won't do anything!
4 + 5 # The rest will be ignored
```

Data and Operations

Data and Operations Definition

When writing a simple program, you can separate out the code into two parts:

the **data** are the values that the program uses to compute new values

the **operations** are the actions that are taken on the values to compute new values

In the expression **2 * 4**, **2** and **4** are the data, and ***** is the operation

Basic Data Types

We'll start out by focusing on two simple data types that Python can use: text and numbers

Text in Python is referred to as **strings**, because it's a string of characters. Text is represented using either double quotes ("foo") or single quotes ('foo').

Numbers in Python come in two types: **ints** (short for integer) and **floats** (short for floating point numbers). Floats can have numbers after the decimal point; ints cannot.

```
# Example strings
"Welcome to programming"
'This should be fun'
```

```
# Example ints
4
-13
```

```
# Example floats
3.14
6.0
```

Showing text

What can we do with text? We can show it to the user!

The **`print()`** command is a built-in action that prints to the interpreter whatever is inside the parentheses. We'll need it to display results when we aren't working directly in the interpreter

Examples:

```
print("Welcome to SAMS")

print(4)

print('Hi ' + 'mom') # We can also concatenate text
```

Exercise 1: Hello World

Go to the schedule page and download the starter file for today's lecture. You'll write exercise code under the comment with the exercise's number.

Exercise 1: write a line of code that prints the text **Hello, World!** to the interpreter.

Congrats- you've written your first program!

Printing multiple things

If we want to print multiple values on the same line, we can separate the values with commas. The values will be printed out separated by spaces.

Example:

```
print("try", "it", "out")
```

Python Math

Python knows how to do all of the math that a calculator can do.

- Addition: +
- Subtraction: -
- Multiplication: *
- Division: /
- Power: **

It can also follow order of operations using parentheses!

```
4 + 16 / 2 # 12
(5 - 1) * 2 # 8
5 ** 2 # 25
```

Exercise 2: Distance

Write a line of code that calculates the distance between the points (1, 2) and (5, 4) and prints it to the interpreter.

Note: You must calculate the distance in the code so that we could update it by replacing (1, 2) and (5, 4) with other points!

Recall that to find the distance between two points, we use the formula

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Advanced Math Operations

In addition to normal operations, we sometimes use **div** and **mod** when programming.

```
5 // 3
```

(div) means 'divide 5 by 3 and cut off the fractional part'.

```
5 % 3
```

(mod) means 'find the remainder of 5 divided by 3'.

Exercise 3: Digits

Write a line of code that calculates the tens-digit of the number 1234, the second digit from the right.

Note: You must calculate the tens-digit such that it would still compute correctly if we replaced 1234 with another integer!

Hint: consider how you could use mod and div to do this...

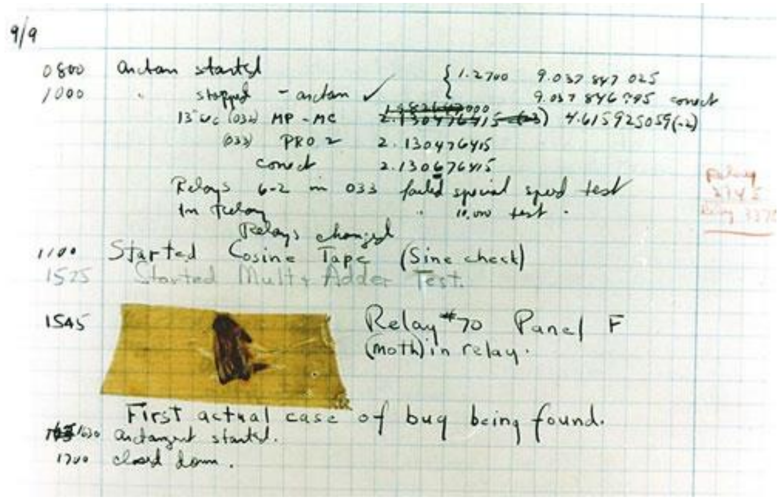
Errors and Debugging

Python Errors

Unlike humans, computers aren't good at improvising. If a single part of a program is written incorrectly, Python won't know what to do. In these situations, Python will show you an **error message**.

A large part of learning how to program involves learning how to read error messages and fix 'bugs' (problems) in the code.

Fun fact: the first program bugs were often actual bugs!



9/9

0800 Anctan started
1000 " stopped - anctan ✓ {1.2700 9.037 847 025
9.037 846 995 conct
13" uc (032) MP - MC ~~1.982147000~~ ~~2.130476415~~ (-3) 4.615925059(-2)
(033) PRO 2 2.130476415
conct 2.130676415
Relays 6-2 in 033 failed special speed test
in relay 11,000 test.
Relays changed
1100 Started Cosine Tape (Sine check)
1525 Started Multy Adder Test.
1545 Relay #70 Panel F
(moth) in relay.
First actual case of bug being found.
1630 Antangent started.
1700 closed down.

Syntax Errors

There are three types of errors you can get while programming. First, **syntax errors** can occur when Python can't parse the code it is given. The code **will not run** until the syntax errors are all fixed.

Examples:

```
four plus six divided by two

print 'Hello World' # missing parantheses

(((5 + 2) * (3 - 4)) # balance your parentheses
```

Runtime Errors

Second, **runtime errors** are errors that Python throws while it is running the code. These generally depend on the values that are being computed, and happen because Python can't perform the stated operation for some reason. The code will run **until the runtime error occurs**.

Examples:

```
3 / (5 - 6 + 1) # We can't divide by zero

Print("Hello World") # Capitalization matters

print("2 + 4 = " + 6) # We can't add strings and numbers
```

Logical Errors

Finally, **logical errors** occur when the code appears to run correctly but gives an incorrect result. **These are the most dangerous errors, because Python won't warn you about them!**

Example:

```
print("2 + 4 = 7")
```

Exercise 4: Error Identification

There are three lines of code under Exercise 3 in the starter file. Write a comment next to each of the lines of code identifying what type of error that code has.

Note: You may want to remove the comment # in front of each line to try running it. Just make sure to add it back in once you're done, so that your code doesn't crash!

```
#print("9 squared =", 9^2)

#print(4 x 7 + 1)

#print("codes" - "s")
```

Debugging

Debugging

Debugging is the process of determining **where** your code is not working correctly, figuring out **why** it is incorrect, and **fixing** the error.

In general, while debugging, the best thing you can do is **read the error messages and code carefully**. However, different errors are best fixed with different approaches.

Remember the three types of errors: **syntax errors**, **runtime errors**, and **logical errors**.

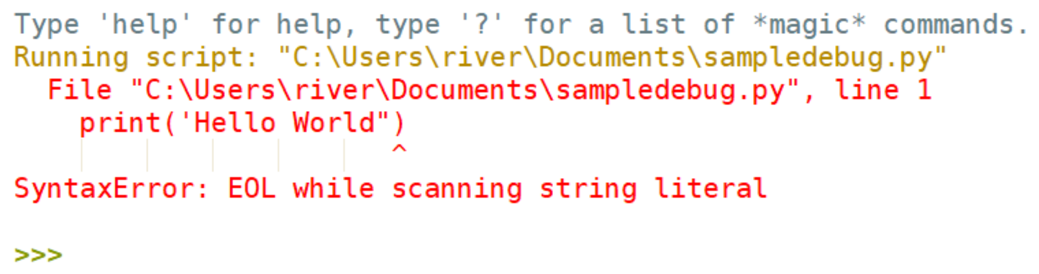
Debugging Syntax Errors

When your program encounters a **syntax error**, follow the following steps:

1. Read the error message and verify that this is a SyntaxError.
2. Look for the line number and the arrow pointing at the code to find the error's location.
3. Carefully read the line of code to find the incorrect syntax.

Example: Syntax Debugging

```
1: print('Hello World")
```



```
Type 'help' for help, type '?' for a list of *magic* commands.
Running script: "C:\Users\river\Documents\sampledebug.py"
  File "C:\Users\river\Documents\sampledebug.py", line 1
    print('Hello World")
                       ^
SyntaxError: EOL while scanning string literal

>>>
```

Debugging Runtime Errors

When your program encounters a **runtime error**, follow the following steps:

1. Read the error message and identify the type of error.
2. Look for the line number, go to that line of code, and identify which part might be associated with the error.
3. Identify how to change the program to achieve the desired outcome.
4. Later on, when we start using variables and state, use print statements to identify where precisely the error occurs.

Example: Runtime Debugging

```
1: print("This is fine")

2: primt("Uh oh")
```

```
Running script: "C:\Users\river\Documents\sampledebug.py"
This is fine
Traceback (most recent call last):
  File "C:\Users\river\Documents\sampledebug.py", line 2, in <module>
    primt("Uh oh")
NameError: name 'primt' is not defined

>>>
```

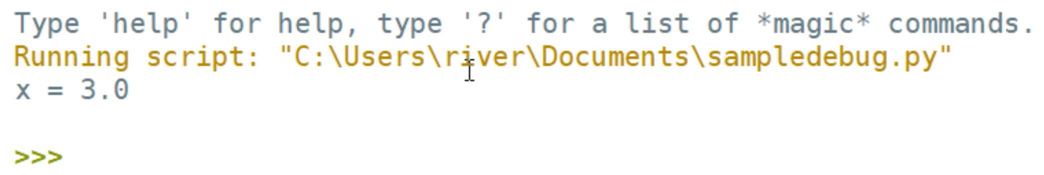
Debugging Logical Errors

When your program encounters a **logical error**, follow the following steps:

1. Determine what the **expected behavior** of the program should be, based on your understanding of the code.
2. Compare the **expected behavior** to the **actual behavior** to see where the two diverge.
3. Narrow the code down to the part where the behavior started diverging, to find the likely source of the bug.
4. Later on, when we start using variables and state, add **print statements** to your code at important junctures to visualize the program's state as it runs.

Example: Logical Debugging

```
# Find the x coordinate where
# 3x + 7 = 5x - 1
print("x =", (7 - 1) / (5 - 3))
```



```
Type 'help' for help, type '?' for a list of *magic* commands.
Running script: "C:\Users\river\Documents\sampledebug.py"
x = 3.0

>>>
```

Exercise 5: Simple Debugging

The following line of code has three errors in it. Uncomment the code, then identify and fix the errors so that the code correctly computes a 15% tip on a meal with three items: a $12 sandwich, a $2 drink, and a $3 desert.

```
#print "Suggested tip:" + 12 + 2 + 3 * 0.15
```

Today's Learning Goals

Understand what a programming language is

Write code that uses numbers, text, and operations to compute simple expressions

Determine when a program isn't working, identify the bug, and fix it

Practice Time!

Exercise Work Time

If you didn't finish any of the exercises during the lecture, try to get them working now!

Have a question? Raise your hand- Prof. Kelly or one of the TAs will help you.

Once you're done with the core exercises, try solving Exercise 6. It's a little trickier than the previous problems...