#1: Building Blocks

SAMS PROGRAMMING C

Course Logistics

Course Website: http://krivers.net/SAMS-m18/

Staff & Student Introductions

Sections A/B vs. C: **we're going to move fast!** We'll have a survey at the end of lecture today to make sure you're in the right section.

Course Plan

Mondays: Lecture, learning new content

Tuesdays: Lab, collaborative guided examples & practice

Thursdays: Homework work time

Week 1: Building blocks (data, functions, conditionals, loops)

Week 2: Algorithmic thinking, testing, and debugging

Week 3: Strings and Lists **(Quiz1 on Thursday)**

Week 4: More Lists and Graphics

Week 5: Input/Output and Animations **<- We'll program Tetris this week!**

Week 6: Coding Metrics **(Quiz2 on Tuesday, no class on Thursday)**

Succeeding in this Course

You will be evaluated based on class participation, quizzes, and homeworks.

- Homeworks are due on **Sundays at 5pm on Autolab**. You can receive help on homeworks during Friday lab and during Instructor and TA office hours (see website). **My office is Gates 4109- come by and visit!**
- Again: if you get stuck, use office hours! Getting help from a TA will save you massive amounts of time
- Quizzes will occur during lab in Week 3 and Week 6

NOTE: homeworks must be completed *individually*. Do not copy code from other students directly (by emailing it) or indirectly (by looking at their screen). You may discuss course ideas at a high (algorithmic, non-code) level to help each other learn, but this must not include discussions of code!

Note 2: This only applies to homeworks and quizzes. During programming practice labs (Tuesdays) while not working on homework problems, feel free to help each other out!

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

Use numbers, text, and boolean values in simple expressions

Write code that stores data using variables and functions

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.

Python Math

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

Examples:

```
4 + 16 / 2
(5 - 1) * 2
```

Advanced Math Operations

```
5 ** 3
```

(pow) means 'raise 5 to the power of 3'

```
5 // 3
```

(div) means 'divide 5 by 3 and cut off the fractional part'. Use it for **step functions**.

```
5 % 3
```

(mod) means 'find the remainder of 5 divided by 3'. Use it for **repeating functions**.

Text in Python

We can also show the user text using Python. Text is represented using either double quotes ("foo") or single quotes ('foo').

The **`print()`** command is a built-in function 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("Hello World!")

print(4 + 5)

print('Hi ' + 'mom')
```

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 Types

Programming languages deal with multiple **data types** which interact in different ways. Numbers can be **int** or **float** types, where ints are integers and floats are decimal (floating point) numbers. Text is a **str** type, short for string (because text is a 'string' of characters).

We can turn numbers into text and text into numbers by using built-in **type-casting functions**. `type()` can be used to find the type of a general value.

Examples:

```
str(4)
int("8")
```

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:

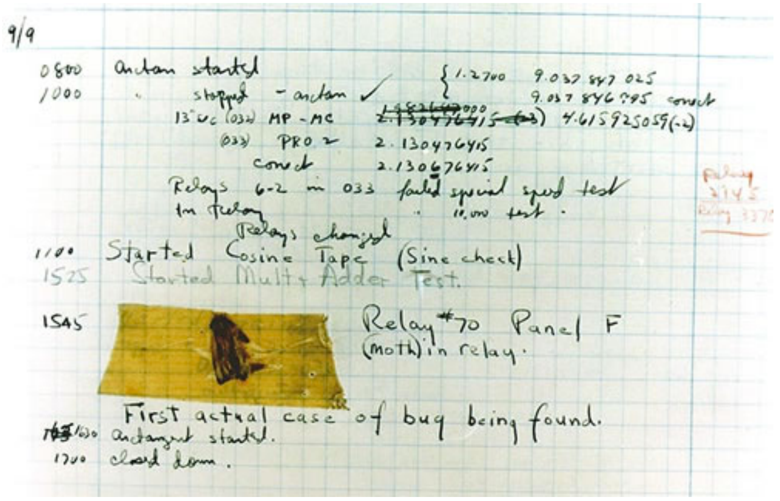
```
# This won't do anything!
4 + 5 # The rest will be ignored
```

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 Antan started
1000 " stopped - antan ✓ {1.2700 9.037 847 025
9.037 846 795 correct
13" UC (032) MP - MC ~~1.982147000~~ 2.130476415 (-3) 4.615925059(-2)
(033) PRO 2 2.130476415
correct 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 Mult + 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

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')
```

```
(((5 + 2) * (3 - 4))
```

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.

Examples:

```
3 / (5 - 6 + 1)
```

```
print("2 + 4 = " + 6)
```

```
int('four')
```

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")
```

Python True/False

We can also **compare values** in Python and evaluate these comparisons as True or False. True and False are **bool** types, which is short for Boolean (named after mathematician George Boole).

Examples:

```
4 < 5

17 == 19

"orange" >= "apple"
```

Combining Booleans

Finally, we can **combine** these boolean values using **logical operations.**

Examples:

```
(21 > 15) and (15 > 5)
(type(4.5) == int) or (type(4.5) == float)
not ("apple" == "orange")
```

Note: comparing floats is dangerous!

Floats don't always behave properly during calculation and comparison...

Examples:

```
(3 + 3 + 3) == 9
```

```
(0.3 + 0.3 + 0.3) == 0.9
```

To fix this, check if the values are **almost equal**. We can use the built-in absolute value function for this.

```
abs((0.3 + 0.3 + 0.3) - 0.9) <= 0.001
```

Built-in Functions

Python has many functions that are built into the language. We've already seen `print` and `abs`, and the type-casting functions. Other useful functions include:

```
len(s) # finds the number of characters in a string
max(4, 6, 2) # finds the maximum of a set of values
min(3, 9, -5) # finds the minimum of a set of values
round(3.14159, 2) # rounds the first number to the second number of digits
assert((1 + 2) == 4) # crashes if the given boolean expression is False
```

Importing Modules

Python has already defined many functions past the builtin ones. These extra functions are organized into different **modules**, which need to be **imported** to be used. Imported functions can then be called with:

```
import <moduleName>
<moduleName>.<functionName>(arguments)
```

Example:

```
import math
print("5! =", math.factorial(5))
```

Storing Data in Variables

Right now, we have no way to store information for use in later expressions. To do this, we have to use **variables**.

A variable is a name that can store a piece of data. The name can be used anywhere where the data would be used normally. We create a variable with the syntax:

```
<varName> = <expression>
```

Example:

```
money = 1.75
quarters = money / 0.25
```

Note: variables can change!

Unlike variables in math, programming variables can change in value during the course of a program.

Prediction Exercise: what value will x hold after each step of the following program?

```
x = 5
y = x * 5
x = y + 3
```

Storing actions in Functions

If we want to reuse a section of code on several different inputs, we can store that code in a **function**.

A function has a name, an input (its **variable parameters**), and an output (the **returned value**).

Functions can be **defined** in code, and they can also be **called** in code.

Defining a Function

When we define a function, we specify how it should work on an **abstracted input**. Defining a function is a bit more complex than the syntax we've learned so far.

```
def <functionName>(<parameters>):
    <functionBody>
    return <functionResult>
```

Note that the function body and return statement are **indented**. Python uses indentation to specify when one or more lines of code should be considered a 'block'. In this case, indentation separates the code that is considered part of the function from any code that follows it.

Defining a Function: Example

Say we want to define a function that converts money into a number of quarters. Our function components are:

Name: `convertToQuarters`

Parameter: `money`

Body: `numQuarters = money / 0.25`

Result: `return numQuarters`

```
def convertToQuarters(money):
    numQuarters = money / 0.25
    return numQuarters
```

Another Type: None

What happens if we don't put a return statement in a function? The function will then return nothing- or, more specifically, the **None value**.

None is a unique built-in value. We can use it to tell the user that there is no viable result.

```
def double(x):
    if type(x) == int:
        return 2 * x
print("double(3):", double(3))
print("double('foo!'):", double('foo!'))
```

Calling a function

Once we have defined a function in a file, we can call it on **specific arguments** to run the code inside the function.

Example:

```
# Assume I have $8.25

print("I have", convertToQuarters(8.25), "quarters")
```

Python will process our code in the convertToQuarters definition using the value 8.25. The number returned by the return statement will then be substituted into the location of the function call.

Note: functions have a different scope

When we define variables inside a function, they **only exist inside the function**. We can't call them in the main code body.

Example:

```
def convertToQuarters(money):

    numQuarters = money / 0.25

    return numQuarters


print(numQuarters * 4) # will crash
```

Exercise: Let's Write a Program

Write a function makes10 that takes two numbers, x and y, and returns True if either the difference between the numbers or the sum of the numbers is equal to 10.

Today's Learning Goals

Understand what a programming language is

Use numbers, text, and boolean values in simple expressions

Write code that stores data using variables and functions

Programming Pretest

See handout. When you're finished, you may hand in your paper and leave.