CSCI-1100 Course Notes

Chuck Stewart, Sibel Adali and Wes Turner

Aug 07, 2018

CONTENTS

ONE

LECTURE 1 — INTRODUCTION

1.1 People

- Professors: Wes Turner, Konstatin Kuzmin

- Instructional Support Coordinator: Erica Eberwein

- TAs and programming mentors: see course website

1.2 Learning Outcomes

1. Demonstrate proficiency in the purpose and behavior of basic programming constructs.

2. Design algorithms and programs to solve small-scale computational programs.

3. Write, test and debug small-scale programs

4. Demonstrate an understanding of the widespread application of computational thinking to real-world problems.

1.3 Textbook

- *Practical Programming: An Introduction to Computer Science Using Python* by Campbell, Gries, and Montojo
    - Available in e-book form
- Very important to get the **second edition**.

1.4 Website and On-Line Resources

- Course notes will be posted at

    http://www.cs.rpi.edu/academics/courses/spring18/csci1100

- Piazza will be used for posting homework assignments and labs, and as a public discussion site:

    https://piazza.com/rpi/spring18/csci1100/home

    You need to sign up with your *rpi.edu* account.

1.5 Other items from the syllabus

- Prof. Turner's office hours are:
  - Tues/Thurs 2:00-3:30 in AE207
- Prof. Kuzmin's office hours are:
  - Tues 4:00-5:30, AE127. Check the webpage.
- Other office hours are posted on-line
- Lab sections are held Tuesdays and Wednesdays. Full credit will be given for finishing the lab within your lab period. Checkpoints completed before your next lab will receive half credit.
- Requirements and grading: lecture exercises, labs, homeworks, tests; letter grades
- Appealing grades
- Class attendance and participation; lecture notes
- Homework late policy:
  - 3 LATE DAYS FOR THE WHOLE SEMESTER
  - 2 LATE DAYS ON ANY ONE ASSIGNMENT
- Academic integrity
- Other exceptions: report to me right now or as soon as you know
- Notes on schedule:
  - Labs started Tuesday. As of now you should have all the pieces you need to successfully complete the remaining labs and homeworks for the semester. If you had problems completing Lab 0, please get help. You have to get it done to proceed.
  - No labs during the weeks of February 19 (President's Day), March 12 (Spring Break) and April 2 (GM Week).
  - Test dates are February 12, March 26, and April 16 all from 6 pm - 7:30 (90 minutes). You must be here or have an excused absence from Student Experience. If you do not have an excused absence for a test, you will get a 0. **No exceptions**!
  - Final exam will be held during finals week. **No exceptions**! So, don't make departure plans until the final exam schedule is posted.

1.6 The Magic of Programming

- Cold, harsh logic, and
- Seemingly primitive syntax. . .
- Leading to soaring creativity!

1.7 Types of Problems We Will Study

- Tools used to help you win at *Words with Friends*
- Image processing and manipulation

- Web searching and crawling

- Programs that work with data from Twitter and Flicker

- Numerical examples, including games and physical simulations

- Perhaps even a simple version of (part of) Pokemon Go.

1.8 Jumping In Quickly with Our First Program — Hello World

- We create a text file called `hello.py` containing just the two lines of Python code:

```python
print('Hello, World!')
print('This is Python')
```

- This is done by launching the Wing IDE, which is specific to creating and running Python programs
    - IDE stands for 'Integrated Development Environment'
        * Two windows will appear — the top being the editor and the bottom being the Python interpreter
- Load the `hello.py` program
- Run it using the interpreter
- We can also type Python code directly into the interpreter.

1.9 Something a Bit More Interesting

- We are going to emphasize computational thinking throughout the semester, so let's look at a fun little problem and get started.

- This problem is posed in *Think Python* and taken from the NPR show *Car Talk*. If you know the answer, do NOT say it!

    "Find the one word in the English language that contains three consecutive double letters."

- We will talk through the steps needed to develop and test a Python program to solve this problem.
    - The file containing this program will be posted on the course website after class.

- We do **not** intend that you will understand the details of the program at this time. Rather, this is just an exercise that illustrates the steps of solving a fun problem computationally.

- On the other hand, it does introduce some elements that will be seeing repeatedly throughout the semester:
    - Files
    - Functions
    - Loops
    - Logic
    - Counting
    - Output
    - Libraries
- In about six weeks, you will understand all parts of this program!

- You can see the code in `three_doubles` from ../class_code

1.10 Looking Back: What Steps Did We Follow?

1. Developing an understanding of what the problem is really asking. This usually involves playing around with small examples.

2. Developing and describing a recipe (an "algorithm") for solving the problem

    - Most recipes will involve multiple parts — multiple functional steps

3. Turning this recipe into a program in the formal language of Python, one of many different programming languages.

    - English is too imprecise for specification of programs.

4. Running this program using the Python interpreter.

1.11 Programs, Compilers, Interpreters, Abstractions

- Python is an interpreted language — run immediately and interactively by the Python interpreter, which is itself another (very complex) program

- Programs in some other (non-interpreted) languages like C, C++ and Java must be compiled (by a "compiler" — another program) into a new program in machine assembly language and then executed.

- In both cases, we write programs that require other programs to run.

    - And, we don't just need just the compiler or interpreter — we need the file system, the operating system, and the command-line interpreter, each of them complicated, multi-part programs themselves.

- We don't really think about the details of these programs; we just think of what they do for us.

    - This is called an "abstraction".

    - It allows us to think about a problem we are trying to solve without thinking about all the details of all the other systems we are depending on.

    - Thinking in terms of abstractions is fundamental to computer science.

1.12 Why Python?

- Python has a very simple syntax

    - The roles of indentation and blank lines cause the most confusion.

- Intepreted languages provide immediate, useful feedback

- Python has a powerful set of tools — abstractions

- Python is widely used in science, engineering and industry.

- Python is good for rapid prototyping

    - Sometimes, after a Python program is written and working, the most time-consuming steps are rewritten in either C or C++ and then integrated with the Python code.

1.13 Two Types of Errors in Our Python Programs

- A *syntax error* is a mistake in the form of the Python code that will not allow it to run.

- A *semantic error* is a mistake in the "meaning" of the program, causing it to produce incorrect output, even if it runs.

- We will demonstrate both types of errors by deliberately introducing errors in our triple double example program.

1.14 Python Versions

- Python, like all programming languages, is continually under development.

- We will be using the latest version, which is 3.6

1.15 Lab 0 — Tuesday and Wednesday!

By the end of Lab 0, you should have:

1. Signed up at the course Piazza web site

   https://piazza.com/rpi/spring2018/csci1100/home

2. Gone to the course page

   http://www.cs.rpi.edu/academics/courses/spring18/csci1100/python_environment.html

   and followed the instructions to install the Python environment on your computer.

   - There are installers for "native" versions of the environment for Windows, Mac OS X and Linux machines.

3. Created a Dropbox account to store back-up copies "in the cloud" of homework and lab solutions and lab for the course.

   - Other cloud-based back-up copies are acceptable.

LECTURE 2 — PYTHON AS A CALCULATOR

2.1 Overview

Most of this is covered in Chapter 2 of *Practical Programming*

- Part 1:
    - Expressions and values
    - Types
    - Precedence
- Part 2:
    - Variables and memory
    - Errors
    - Typing directly in the interpreter vs. running programs; use of *print*
    - Documentation and variable names

Throughout we will pay attention to problems and mistakes, both real and potential.

2.2 Aside: Lectures, Note Taking and Exercises

- Lecture notes are outlines.
    - Therefore, you should hand-write details as we cover them in class.
- We will create and run examples in class.
    - You should write down the shorter ones
    - We will post the longer ones on-line, but you should write down as much as you can, especially the results of running the examples. If you don't know what to expect, you can't debug errors.

2.3 Python As a Calculator

We will start class by using Python as an interactive calculator, working through a number of examples.

- The area of a circle.
- The number of minutes in a year.

- The volume of a box.

- The volume of the earth in cubic kilometers.

- In doing so, we will look at the basic operations of +, -, *, /, and ** (exponentiation).

2.4 Whole Number Calculations

- Most of the calculations in the foregoing examples involve numbers with fractional values.

- Sometimes we want to use whole number of calculations, for example to convert a number of minutes to a number of hours and minutes (less than 60).

- In this case, Python offers us different forms of divisions

    - // is the operator for whole number division

    - % is used to find the remainder

- For this kind of division, be careful of negative numbers. Can you figure out how they work with // and %?

    - This is a case where experimentation with the Python interpreter is crucial. We will encourage you to do this throughout the semester.

2.5 Python Types

- We have seen what look like real numbers and whole numbers. These are two different *types* in Python.

- A type is a set of possible values — also called a "representation" — and a set of operations on those values.

    - Our first examples are the "operators" such as +, -, *, / and **

- Common Python types we are working with initially include `float`, `int` (short for integer) and `str` (short for string)

- Each value we create will be referred to as an *object*

2.6 float

- The `float` type approximates real numbers. But computers can't represent all of them because computers only dedicate a finite amount of memory to each.

- Limited precision: we'll look at 2/3, 5/3 and 8/3

- Any time integers and floats are mixed and any time we apply division, the result is always a float.

2.7 int

- The `int` (integer) type is analogous to whole numbers:

```
{ ..., -4, -3, -2, -1, 0, 1, 2, 3, 4, ...}
```

- Python can represent seemingly arbitrarily large integers, which is not true of other languages like C, C++ and Java

– Using the Python interpreter, we will look at the examples of:

```
>>> 1**1**1
>>> 2**2**2
>>> 3**3**3
```

2.8 Precedence

- Consider the formula for converting 45 degrees Fahrenheit to Celsius. What is wrong with the following computation?

```
>>> 45 - 32 * 5 / 9
```

- Largely following the standard rules of algebra, Python applies operations in order of *precedence*. Here is a summary of these rules from highest to lowest:

    1. ( ) - parentheses

    2. ** - the exponentiation operator, ordered *right-to-left*

    3. - - the negation (unary minus) operators, as in -5**2

    4. *, /, //, % - ordered *left-to-right*

    5. +, - - ordered *left-to-right*

- This example also suggests an important question: how do we know we've made a mistake - introduced a *bug* - in our code?

2.9 Part 1 Practice Problems

1. The consider the following evaluations (some of them trivial). Which results are `float` objects and which are `int` objects?

```
>>> 9          # 1
>>> 9.         # 2
>>> 9.0        # 3
>>> 9 + 3      # 4
>>> 9 - 3.     # 5
>>> 9 / 4      # 6
>>> 9 // 4     # 7
>>> 9. // 4    # 8
```

2. What is output by the Python interpreter?

```
>>> 2**3**2
>>> (2**3)**2
>>> -2**3 - 2 * 5
```

3. Write a single line of Python code that calculates the radius of a circle with area 15 units and prints the value. The output should just be the number that your code produces. Your code should include the use of an expression involving division and exponentiation (to compute the square root). Use the value 3.14159 for *pi*.

2.10 Part 2

2.11 Variables and Assignment

- Most calculators have one or several memory keys. Python, and all other programming languages, use "variables" as their memory.

- We'll start with a simple example of the area of a circle, typed in class. You will notice as we go through this that there is no output until we use the *print* functions.

- Here is a more extensive example of computing the volume and surface area of a cylinder:

```
>>> pi = 3.14159
>>> radius = 2
>>> height = 10
>>> base_area = pi * radius ** 2
>>> volume = base_area * height
>>> surface_area = 2 * base_area + 2 * pi * radius * height
>>> print("volume is", volume, ", surface area is", surface_area)
volume is 125.6636 , surface area is 150.79632
```

- A variable is a name that has a value "associated" with it.

  – There are six variables in the above code.

- The value is substituted for the variable when the variable appears on the right hand side of the =.

- The value is **assigned to** the variable when the variable name appears on the left hand side of the =.

2.12 More on Variable Assignment

- The operator = is an assignment of a value (calculated on the right side) to a variable (on the left).

- In the following..

```
>>>  base_area = pi * radius ** 2
```

  Python

  – accesses the values associated with the variables `pi` and `radius`,

  – squares the value associated with `radius` and then multiplies the result by the value associated with the variable `pi`,

  – associates the result with the variable `base_area`

- Later, Python accesses the value of `base_area` when calculating the values to assign to `volume` and `surface_area`.

- Thus, the meaning of = in Python is quite different from the meaning of = in mathematics.

- The statement

```
>>>  base_area * height = volume
```

  is not legal Python code. Try it!

- It takes a while to get accustomed to the meaning of an assignment statement in Python.

2.13 print

- Consider the line

```
>>> print("volume is", volume, ", surface area is", surface_area)
```

- `print` is a Python "function" that combines *strings* (between the quotations) and values of variables, separated by commas, to generate nice output.

- We will play with a number of examples in class to illustrate use of `print`. As the semester progresses we will learn a lot more about it.

2.14 Variable Names

- Notice that our example variable names include letters and the `_` (underscore) character.

- Legal variable names in Python must

  – Start with a letter or a `_`, and

  – Be followed by any number of letters, underscores or digits.

  Characters that are none of these, including spaces, signal the end of a variable name.

- Capital letters and small letters are different

- We will look at many examples in class.

2.15 Putting Your Code in a File

- So far in today's lecture we have written our code using the *Python Shell*.

  – This sends your Python statements directly to the *interpreter* to execute them

- Now we will switch to writing and saving our code in a file and then sending the file to the Python interpreter to be run.

- We will demonstrate using the surface area and volume calculations from earlier in lecture.

- Almost all code that you write for lecture exercises, labs, and homework assignments will be stored in files.

- You will practice in Lab 1 next week.

- Sometimes in class we will still type things directly into the shell. You will know we are doing this when you see >>>

2.16 Syntax and Semantic Errors

- Python tells us about the errors we make in writing the names of variables and in reversing the left and right side of the = operator.

- These are examples of *syntax errors* — errors in the form of the code.

- Programs with syntax errors will not run; the Python interpreter inspects the code and tells us about these errors before it tries to execute them. We can then fix the errors and try again

- More difficult to find and fix are *semantic errors* — errors in the logical meaning of our programs resulting in an incorrect result.

    - We have already seen an example of a semantic error. Can you think where?

    - Throughout the semester we will discuss strategies for finding and fixing semantic errors.

2.17 Python Keywords

- All variable names that follow the above rules are legal Python names *except* for a set of "keywords" that have special meaning to Python.

- Keywords allow us to write more complicated operations — involving logic and repetition — than just calculating.

- You can get a list of Python keywords by typing into the shell

```
>>> import keyword
>>> print(keyword.kwlist)
```

- Over the next few lectures, we will soon understand the detailed meaning of the . in the above statement.

2.18 Do Variables Exist Before They Are Assigned a Value?

- Suppose we forgot to assign `pi` a value? What would happen?

    - Try it out!

- Variables do not exist until they are assigned a value.

- This is a simple form of semantic error.

2.19 Example to Consider

1. Create 2 invalid variable names and 4 valid variable names from the _ character, the digit 0, and the letter `a`.

2.20 Mixed Operators

- Assignments of the form

```
>>> i = i + 1
```

are commonly seen in Python. We will take a careful look at what is happening here.

- Python contains a short-hand for these:

```
>>> i += 1
```

These two statements are exactly equivalent.

- Other mixed operators include

```
 -=      *=     /=
```

but += is used most commonly for reasons that will gradually become clear over the first half of the semester.

2.21 Terminology: Expressions

- Expressions are formed from combinations values, variables and operators.

- In the examples we've seen so far, expressions are on the right-hand side of an assignment statement, as in:

```
>>> surface_area = 2 * base_area + 2 * pi * radius * height
```

2.22 Part 2 Practice Problems

1. Which of the following are legal Python variable names?

```
import
56abc
abc56
car-talk
car_talk
car talk
```

2. Which of these lines of code contain syntax errors? Once you fix the syntax errors, the program (assume this has been typed into a file and run in the Wing IDE) will still not correctly print the area of a circle with radius 6.5. What two more changes are needed to fixed these errors?

```
pi = 21 // 7
area = pi * r * r
r = 6.5
r + 5 = r_new
print(area)
```

3. Assuming you start with $100 and earn 5% interest each year, how much will you have at the end of one year, two years and three years? Write Python expressions to calculate these, using variables as appropriate. We will write the solution into a file and run the file using the interpreter.

4. What is the output of the following Python code (when typed into a file and run in the interpreter)? Try to figure it out by hand before typing the statements into a file and running in the Python interpreter.

```
x = 12
y = 7.4
x -= y
print(x, y)
y = y-x +7
z = 1
x *= 2 + z
print(x, y)
x += x*y
print(x, y)
```

2.23 Summary — Important Points to Remember

- Expressions are formed from combinations of values, variables and operators
- Values in Python are one of several different types — integers, floats, and strings for now.
- Variables are Python's form of memory
- Python keywords can not be used as variables.
- = is Python's means of assigning a value to a variable
- Variables do not exist in Python until they are given a value
- Make sure you have the precedence correct in your Python expressions.

LECTURE 2 — EXERCISES

3.1 Overview

These problems are exercises to work on at the end of lecture and submit for a small part of your grade. Normally, solutions will be due within 24 hours of the end of the lecture they are associated with. For these Lecture 2 exercises only, they will be due by the end of Lab 1 so that any problems with submission can be ironed out.

Students are welcome to work on these problems in small groups, but each student should write the final version of their solutions independently. Each student must submit their own solutions.

3.2 Getting Started with the Wing IDE

Open up the Wing IDE:

- You can practice with small sections of Python code by typing in the interpreter in the lower right pane.

- In order to create a Python program that you save to a file, click *File -> New*. You can save it to a file by typing *File -> Save As*

- As discussed in Lab 0 you should save your programs in an organized manner within your Dropbox folder.

- Once you have drafted your code to solve a problem or, better yet, have written enough that you are ready to experiment with what you have, click on the green triangle to run your code. You will see the results in the interpreter pane on the lower right.

- If you do not see the green triangle, you need to save your code to a file first.

Now you are ready to proceed…

3.3 Problems for Grade Submission

1. Write a single line of Python code that converts the temperature 64 from Celsius to Fahrenheit and prints the value. Submit a Python file containing just this single line of code. The output should just be the number that your code produces. Your code must include the use of an expression involving multiplication and a `print` function call.

2. Write Python code that creates three variables called `length`, `width` and `height` to store the dimensions of a 16.5 x 12.5 x 5 box. Write additional code that calculates the volume of the box and calculates its surface area, storing each in a variable. Print the values of these variables. Your code must use five assignment statements and two print function calls. Submit a file containing these seven lines of Python code. Your output should be

```
volume = 1031.25
area = 702.5
```

3. Your problem is to determine the output of the Python program shown below. You must submit a text file showing the output. (Hint: there should be two lines with one integer on each line.) While it is possible to just run the program and copy the output, we *strongly* encourage you to not do this. You need to develop the ability to read code and understand what it will do. You will be tested on it.

```
z = 2
z = z**2**3
print(z)
x = 6
x = x**2 + 6 - z // 10 * 2
print(x)
```

LECTURE 3 — PYTHON STRINGS

4.1 Reading

This material is drawn from Chapter 4 of *Practical Programming*, 2nd edition.

4.2 More Than Just Numbers

- Much of what we do today with computers revolves around text:
    - Web pages
    - Facebook
    - Text messages

  These require working with *strings*.

- Strings are our third type, after integers and floats.

- We've already seen the use of strings in output,

```python
print("Hello world")
x = 8
y = 10
print("Value of x is", x, "value of y is", y)
```

4.3 Topics for Today

- String basics
- String operations
- Input to and output from your Python programs

4.4 Strings — Definition

- A string is a sequence of 0 or more characters delimited by single quotes or double quotes.

```
'Rensselaer'
"Albany, NY"
'4 8 15 16 23 42'
''
```

- We can print strings:

```
>>> print("Hello, world!")
Hello, world!
```

- Strings may be assigned to variables:

```
>>> s = 'Hello'
>>> t = "Good-bye"
>>> print(s)
Hello
>>> t
'Good-bye'
```

- Notice that unlike integers and floats there is now a difference between asking the Python function `print` to output the variable and asking the Python interpreter directly for the value of the variable.

4.5 Combining Single and Double Quotes in a String

- A string that starts with double quotes must end with double quotes, and therefore we can have single quotes inside.

- A string that starts with single quotes must end with single quotes and therefore we can have double quotes inside.

- To illustrate this, we will take a look at

```
>>> s = 'He said, "Hello, World!"'
>>> t = "Many single quotes here '''''''  and here ''' but correct."
```

4.6 Multi-Line Strings

- Ordinarily, strings do not extend across multiple lines, causing an error if you try.

- But, starting and ending a string `"""` or `'''` tells Python to allow the string to cross multiple lines.

  - Any character other than `'''` (or `"""`, if that is how the string started) is allowed inside the string.

- Example,

```
>>> s1 = """This
is a multi-line
string."""
>>> s1
'This\nis a multi-line\nstring.'
>>> print s1
This
is a multi-line
string.
>>>
```

- Notice the \n when we ask Python for the value of the string (instead of printing it). This is an *escape character*, as we will discuss next.

4.7 Escape Characters

- Inserting a \ in the middle of a string tells Python that the next character will have special meaning (if it is possible for it to have special meaning).

- Most importantly:

    - \n — end the current line of text and start a new one

    - \t — skip to the next "tab stop" in the text. This allows output in columns

    - \' — do not interpret the ' as a string delimiter

    - \" — do not interpret the " as a string delimiter

    - \\ — put a true back-slash character into the string

- We'll explore the following strings in class

```
>>> s0 = "*\t*\n**\t**\n***\t***\n"
>>> s1 = "I said, \"This is a valid string.\""
```

4.8 String Operations — Concatenation

- Concatenation: Two (or more) strings may be concatenated to form a new string, either with or without the + operator. We'll look at

```
>>> s0 = "Hello"
>>> s1 = "World"
>>> s0 + s1
>>> s0 + ' ' + s1
>>> 'Good' 'Morning' 'America!'
>>> 'Good ' 'Morning ' 'America!'
```

- Notice that

```
>>> s0 = "Hello"
>>> s1 = " World"
>>> s0 s1
```

is a syntax error but

```
>>> "Hello" " World"
```

is not. Can you think why?

4.9 String Operations — Replication

- You can replicate strings by multiplying them by an integer:

```
>>> s = 'Ha'
>>> print(s * 10)
HaHaHaHaHaHaHaHaHaHa
```

- What do you think multiplying a string by a negative integer or 0 does? Try it.

- Many expressions you might try to write involving strings and either ints or floats are illegal Python, including the following:

```
>>> 'Hello' * 8.1
>>> '123' + 4
```

Think about why

4.10 Practice Problems - Part 1

We will go over these during lecture:

1. Which are valid Python strings:

```
>>> s1 = '"Hi mom", I said.  "How are you?"'
>>> s2 = '"Hi mom", I said.  '"How are you?"
>>> s3 = '"Hi mom", I said.  '"How are you?"'
>>> s4 = """'Hi mom", I said.  '"How are you?"'"""
>>> s5 = ""I want to be a lion tamer!"'
>>> s6 = "\"Is this a cheese shop?\"\n\t'Yes'\n\t\"We have all kinds!\""
```

For those that are not valid, what needs to be fixed? For those that are, what is the output when they are passed to the `print` function?

2. What is the output?

```
>>> s = "Cats\tare\n\tgood\tsources\n\t\tof\tinternet\tmemes"
>>> s
>>> print(s)
```

3. What is the output?

```
print('\\'*4)
print('\\\n'*3)
print('Good-bye')
```

4. Which of the following are legal? For those that are, show what Python outputs when these are typed directly into the interpreter.

```
>>> 'abc' 'def'
>>> 'abc' + 'def'
>>> 'abc ' + 'def'
>>> x = 'abc'
>>> y = 'def'
>>> x+y
>>> x y
>>> s1 = 'abc'*4
>>> s1
>>> s2 = 'abc '*4
>>> print(s2)
```

4.11 String Operations — Functions

- Python provides many operations for us to use in the form of **functions**. We have already seen `print()`, but now we are going to look at other functions that operate on strings.

- You can compute the length of a string with `len()`.

```
>>> s = "Hello!"
>>> print(len(s))
```

- Here is what happens:

    1. Function `len` is provided with the value of the string associated with variable `s`

    2. `len` calculates the number of characters in the provided string using its own code, code that is *built-in* to Python.

    3. `len` *returns* the calculated value (in this case, 6) and this value is sent to the `print` function, which actually generates the output.

- We will learn more about using functions in Lectures 4 and 5.

4.12 Example String Functions

- We will look at examples of all of the following during lecture...

- You can convert an integer or float to a string with `str()`.

- You can convert a string that is in the form of an integer to an integer using `int()`

- You can convert a string that is in the form of a float to a float using, not surprisingly, `float()`

4.13 The print Function in More Detail

- We already know a bit about how to use `print()`, but we can learn about more using `help()`

```
help(print)
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

- `flush` is useful when trying to debug. If you are trying to trace your program execution using print, adding `flush=True` as your final argument will give you more accurate results. We will talk about this more later.

- For now, we will focus on the `sep` and `end` and illustrate with examples.

4.14 User Input

- Python programs can ask the user for input using the function called `input`.

- This waits for the user to type a line of input, which Python reads as a string.

- This string can be converted to an integer or a float (as long as it is properly an int/float).

- Here is a toy example

```python
print("Enter a number")
x = float(input())
print('The square of', x, 'is', x*x)
```

- We can also insert the string right into the `input` function call:

```python
x = input("Enter a new number ")
x = float(x)
print('The square of', x, 'is', x*x)
```

- A similar function exists to convert a string to an integer:

```python
x = input("Enter an integer ")
x = int(x)
```

- We will use this idea to modify our area and volume calculation so that the user of the program types in the numbers.

  - The result is more useful and feels more like a real program (run from the command line).

  - It will be posted on the course website.

4.15 Practice Problems – Part 2

1. What is the output for this Python program?

```python
print(len('George'))
print(len(' Tom  '))
s = """Hi
sis!
"""
print(len(s))
```

2. Which of the following are legal? For those that are, show what Python outputs when these are typed directly into the interpreter.

```python
>>> 'abc' + str(5)
>>> 'abc' * str(5)
>>> 'abc' + 5
>>> 'abc' * 5
>>> 'abc' + 5.0
>>> 'abc' + float(5.0)
>>> str(3.0) * 3
```

3. What is the output of the following when the user types 4 when running the following Python program?

```
x = input('Enter an integer ==> ')
x = x*2
x = int(x)
x *= 2
print("x is:", x)
```

4. What is the output when the user types the value 64 when running the following Python program?

```
x = input('Enter an integer ==> ')
y = x // 10
z = y % 10
print(x, ',', y, z, sep='')
```

What happens when you do not have the call to the `int` function?

5. Write a program that requests an integer from the user as an input and stores in the variable `n`. The program should then print `n` 1's with 0's inbetween. For example if the user input the value 4 then the output should be

```
1010101
```

4.16 Summary

- Strings represent character sequences — our third Python type
- String operations include addition (concatenate) and replication
- Functions on strings may be used to determine length and to convert back and forth to integers and floats.
- Escape sequences change the meaning of special Python characters or make certain characters have special meaning.
- Some special characters of note: \n for new line, \t for tab. They are each preceded by \
- The `print()` function offers significant flexibility.
- We can read input using `input()`

LECTURE 3 — EXERCISES

5.1 Overview

Solutions to the problems below must be sent to Submitty for grading. A separate file must submitted for each problem, as practiced in Lab 1. For these Lecture 3 exercises, you must submit your solutions before Wednesday, September 13 at 11:59:59. Starting with Lecture 4, you will be required to submit you lecture exercises within 24 hours of the end of lecture. Students are welcome to work on these problems in small groups, but each student should write the final version of their solutions independently to assure themselves that they understand.

5.2 Problems

1. Which of the following are valid strings? Upload a text file to Submitty that contains just the variable names that are assigned to strings that are correct. For example if only the first two were correct your file would contain s0 on the first line and s1 on the second.

   ```
   >>> s0 = "Sheldon Cooper's apartment is in Pasedena"

   >>> s1 = 'This cheese shop's cheese is all gone"

   >>> s2 = """We are
   "The Knights of the Round Table"
   """

   >>> s3 = "Toto, I said,\n"We aren't in Kansas, anymore!"


   >>> s4 = 'Have you seen the "Final Five"'s picture?'


   >>> s5 = "Have you seen the 'Final Five''s picture?"
   ```

2. Submit a Python file that includes a single line of code that prints 25 '*' characters followed by 25 '+' characters, with no space in between. It must, of course, use the `print` function. The two characters must appear in your code *much less* than 25 times - at most three each!

3. Write a program that assigns the value 4 to the variable x, the value 2 to the variable y and then uses exactly *three* `print` function calls to generate the output below (four lines, with the second line blank). The `print` calls must use the variables x and y rather than the values 4 and 2. The trick is to change the assignment of the sep and end parameters in the call to `print`. The character 4 is the first character on the 1st, 3rd and 4th lines of output.

```
4 2

4,2
42
```

LECTURE 4 — USING FUNCTIONS AND MODULES

6.1 Reading

- Material for this lecture is drawn from Sections 3.1, 6.1 and 7.1-7.3 of *Practical Programming*.
- Topics that we will discuss include:
    - Python functions for different data types.
    - String functions and calling functions on objects
    - Using modules provided with Python
- We will revisit all these concepts several times throughout the semester.

6.2 What have we learned so far?

- So far, we have learned about three basic data types: integer, float and strings.
- We also learned some valuable functions that operate on strings (`len`) and that convert between data types (`int`, `str`, `float`)

```
>>> name = "Pickle Rick"
>>> len(name)
11
```

- The functions that Python provides are called *built-in* functions.
- We will see examples of these functions and experiment with their use in this class.

6.3 How about numerical functions?

- Many numerical functions also exist. Let us experiment with some of these first. You should make a note of what they do.
    - `abs()`
    - `pow()`
    - `int()`
    - `float()`
    - `round()`

   – `max()`

   – `min()`

- We will look carefully in class at how these work.

6.4 Objects and Methods

- All variables in Python are objects.

- Objects are abstractions:

   – Each object defines an organization and structure to the data they store.

   – They have operations/functions — we call them **methods** — that we can apply to access and manipulate this data.

   – We don't think about how they are implemented; instead we just think about how to use them. This is why they are called abstractions.

- Methods associate with objects often use a function call syntax of the form

```
variable.method(arguments)
```

For example:

```
>>> b = 'good morning'
>>> b.find('o', 3)
```

This also works on particular values instead of variables:

```
value.method(arguments)
```

as in

```
>>> 'good morning'.find('o', 3)
```

- You can see all the methods that apply to an object type with help as well. Try:

```
>>> help(str)
```

6.5 String Methods

- Here are a few more (of many) string methods:

```
>>> name = "Neil Degrasse Tyson"
>>> name.lower()
'neil degrasse tyson'
>>> lowername = name.lower()
>>> lowername.upper()
'NEIL DEGRASSE TYSON'
>>> lowername.capitalize()
'Neil degrasse tyson'
>>> lowername.title()
'Neil Degrasse Tyson'
```

```
>>> "abracadabra".replace("br", "dr")
'adracadadra'
>>> "abracadabra".replace("a", "")
'brcdbr'
>>> "Neil Degrasse Tyson".find(" ")
4
>>> "Neil Degrasse Tyson".find("a")
9
>>> "Neil Degrasse Tyson".find("x")
-1
>>> "Monty Python".count("o")
2
>>> "aaabbbfsassassaaaa".strip("a")
'bbbfsassass'
```

- As described above, all of these are called in the form of `object.method(arguments)`, where `object` is either a string variable or a string value.

- Not all functions on objects are called this way. Some are called using more of a function form, while others are called as operators.

```
>>> episode = "Cheese Shop"
>>> episode.lower()
'cheese shop'
>>> len(episode)
11
>>> episode + "!"
'Cheese Shop!'
```

  – We will see the reason for the differences later in the semester.

- Note of caution: none of these functions change the variable that they are applied to.

6.6 Practice Problems (1)

1. Write code that takes a string in a variable called `phrase` and prints the string with all vowels removed.

2. Create a string in a variable and assign it to a variable called `name`. Write code to create a new string that repeats each letter `a` in `name` as many times as `a` appears in `name` (assume the word is all lower case).

   For example,

```
>>> name = "amos eaton"
## your code goes here



>>> name
'aamos eaaton'
```

3. Given a string in a variable called `name`, switch all letters `a` and `e` (only lowercase versions). Assume the variable contains only letters and spaces.

   Hint: first replace each 'a' with '1'.

```
>>> name = "Rensselaer Polytechnic Institute"
## your code goes here


>>> name
'Ranssalear Polytachnic Instituta'
```

6.7 String Format Method

- The `format()` method provides a nice way to produce clean looking output.

- For example, consider the code

```
>>> pi = 3.14159
>>> r = 2.5
>>> h = 10**0.5
>>> volume = pi * r**2 * h
>>> print('A cylinder of radius', r, 'and height', h, 'has volume', volume)
A cylinder of radius 2.5 and height 3.1622776601683795 has volume 62.09112421505237
```

- Now look at what we can do with the `format()` method:

```
>>> out_string = 'A cylinder of radius {0:.2f} and height {1:.2f} has volume {2:.2f}'.
↪format(r, h, volume)
>>> print(out_string)
A cylinder of radius 2.50 and height 3.16 has volume 62.09
```

- Method `format()` replaces the substrings between {} with values from the argument list.

  - {0:.2f} means argument 0, will be formatted as a float with 2 digits shown to the right of the decimal place.

    * Notice it applies rounding.

  - We can leave off the 0, the 1, and the 2 from before the : unless we want to change the order of the output.

  - We can leave off the :.2f if we want to accept print's normal formatting on float outputs.

- There are many variations on this and we will see quite a few as we progress through the semester.

6.8 Built-In Functions

- All the functions we have seen so far are *built-in* to the core Python. It means that these functions are available when you start Python.

- Type

```
>>> help(__builtins__)
```

to see the full list.

6.9 Modules

- Now we will begin to look at using functions that are not built into the core of Python but rather imported as **modules**.

- Modules are collections of functions and constants that provide additional power to Python programs.

- Some modules come with Python, but are not loaded automatically. For example the `math` module.

- Other modules need to be installed first. When we installed software in Lab 0, we installed a library called `pillow` that has a number of image manipulation modules.

- To use a function in a module, first you must load it into your program using `import`. Let's see the `math` module:

```
>>> import math
>>> math.sqrt(5)
2.2360679774997898
>>> math.trunc(4.5)
4
>>> math.ceil(4.5)
5.0
>>> math.log(1024,2)
10.0
>>> math.pi
3.141592653589793
```

- We can get an explanation of what functions and variables are provided in a module using the `help` function

```
>>> import math
>>> help(math)
```

6.10 Different Ways of Importing

- The way you import a module determines what syntax you need to use the contents of the module in your program.

- We can import only a selection of functions and variables:

```
>>> from math import sqrt,pi
>>> pi
3.141592653589793
>>> sqrt(4)
2.0
```

- Or we can give a new name to the module within our program:

```
>>> import math as m
>>> m.pi
3.141592653589793
>>> m.sqrt(4)
2.0
```

- Both of these methods help us distinguish between the function `sqrt` and the data `pi` defined in the math module from a function with the same name (if we had one) in our program.

- We can also do this (which is NOT recommended!):

```
>>> from math import *
```

Now, there is no name difference between the math module functions and ours. Since this leads to confusion when the same name appears in two different modules it is almost always avoided.

6.11 Program Structure

- We have now seen several components of a program: `import`, comments, and our own code, including input, computation and output statements. We will add more components, such as our own functions, as we proceed through the semester.

- You should organize these components in your program files to make it easy to see the flow of the program

- We will use the following convention to order the program components:

  - an initial comment explaining the purpose of the program,

  - all `import` statements,

  - then all variables and input commands,

  - then all computation,

  - finally all output.

6.12 Putting It All Together

- In the rest of this class we will write a program that first asks the user for a name, then asks for the radius and height of a cylinder, and finally prints the surface area and volume of the cylinder, nicely formatted.

6.13 Practice Problems (2)

1. The `math` module contains the constant `e` as well as `pi`. Write code that prints these values accurate to 3 decimal places and then write code that computes and outputs

$$\pi^e$$

and

$$e^\pi$$

both accurate to 2 decimal places.

2. Write a short program to ask the user to input height values (in cm) three times. After reading these values (as integers), the program should output the largest, the smallest and the average of the height values.

3. What happens when we type

```
import math
math.pi = 3
```

and then use `math.pi`?

6.14 Summary

- Python provides many functions that perform useful operations on strings, integers and floats.

- Some of these functions are *built in* while others are organized into modules.

- Be aware of the differences between how functions are called. You must remember them to call them correctly.

  – Functions that require dot notation, applying the function to an object (or a variable containing an object):

  ```
  >>> "abc".upper()
  'ABC'
  ```

  – Functions that are called with arguments (no dot notation):

  ```
  >>> x = -4.6
  >>> abs(x)
  4.6
  >>> round(x)
  -5
  ```

  Note that these functions are actually aliases. The same function also exists in dot notation.

  ```
  >>> x = -4.6
  >>> x.__abs__()
  4.6
  >>> x.__round__()
  -5
  >>> 4.6.__round__()
  5
  >>> (-1).__abs__()
  1
  ```

- After a module is imported, the functions in the module can be used by a call of the form:

  ```
  module_name.function_name(arguments)
  ```

- You can see the details of a function by:

  ```
  >>> help(module_name.function_name)
  ```

- Python has many modules that make it easy to do complicated tasks. If you do not believe it, try typing:

  ```
  >>> import antigravity
  ```

LECTURE 4 — EXERCISES

Solutions to the problems below must be sent to Submitty for grading. A separate file must submitted for each problem. These must be submitted by 4 pm on Tuesday, January 30.

1. Write a program that assigns a string to the variable called `phrase` and then transforms `phrase` into a hashtag. In other words, all words in `phrase` are capitalized, all spaces are removed, and a # appears in front. Store the result in a variable called `hashtag`. Then print the value of both `phrase` and `hashtag`. Your program should start with

```
phrase = 'Things you wish you knew as a freshman'
```

and the output from the program (using `print()` function calls) should be

```
The phrase "Things you wish you knew as a freshman"
becomes the hashtag "#ThingsYouWishYouKnewAsAFreshman"
```

Note that the output includes the quotation marks.

2. One of the challenges of programming is that there are often many ways to solve even the simplest problem. Consider computing the area of the circle with the standard formula

$$a(r) = \pi r^2$$

Fortunately, we now have `pi` from the *math* module, but to compute the square of the radius we now can use `**` or `pow()` or we can just multiply the radius times itself. To print the area accurate to only a few decimal places we can now use the string `format()` method or the `round()` built-in function, which includes an optional second argument to specify the number of decimal places.

Write a short Python program that computes and prints the areas of two circles, one with radius 5 and the other with radius 32. Your code must use `**` once and `pow()` once and it must use `format()` once and `round()` once. The output should be exactly

```
Area 1 = 78.54
Area 2 = 3216.99
```

LECTURE 5 — PYTHON FUNCTIONS

8.1 Reading

Most of this is covered late in Chapter 3 of *Practical Programming*.

8.2 Why Functions?

- The purpose of today's class is to introduce the basics of writing and running Python functions.
  - Repeating code is painful.
  - It is also hard to distinguish between the same code repeated three times and three different computations.
  - It is easy to find a mistake in one copy of a section of code and forget to fix it in the other copies.
- Learn a programmer's motto: DRY – don't repeat yourself.
  - Define it once and use it multiple times.
- Functions are extremely useful for writing complex programs:
  - They divide complex operations into a combination of simpler steps.
  - They make programs easier to read and debug by abstracting out frequently repeated code.

8.3 Functions

- As we have learned, a function
  - Takes as input one or more arguments.
  - Computes a new value, a string or a number.
  - Returns the value, so that it can be assigned to a variable or output.
- Let's recall this with a built-in function:

```
>>> len('RPI Puckman')
11
```

Can you identify the input argument, the computation and the returned value?

8.4 A Function to Compute the Area of a Circle

- In mathematics, many functions are given as formulas. You might write a function to calculate the area of a circle as

$$a(r) = \pi r^2$$

- In Python, typing into a file (in the upper pane of the Wing IDE), we write

```python
def area_circle(radius):
    pi = 3.14159
    area = pi * radius**2
    return area
```

- Note that the `def` is not indented and the other lines are indented four spaces.

- We add unindented code the file below the definition of `area_circle()` to execute the function and calculate the area:

```python
a = area_circle(1)
print(a)
print('A circle with radius 2 has area {:.2f}'.format(area_circle(2)))
r = 75.1
a = area_circle(r)
print("A circle with radius {:.2f} has area {:.2f}".format(r,a))
```

Note that by using examples with small values for the radius we can easily check that our function is correct.

- Important syntax includes

  – Use of the keyword `def` and the `:` to indicate the start of the function

  – Indentation for the lines after the `def` line

  – Unindented lines for code outside the function to indicate the end of the function.

8.5 What does Python do when we run this code?

We can visualize this using the website `http://www.pythontutor.com`

1. Reads the keyword `def` and notes that a function is being defined.

   - The line that starts with `def` is called the function *header*.

2. Reads the rest of the function definition, checking its syntax.

3. Notes the end of the definition when the unindented code is reached.

4. Sees the function call inside the assignment statement

```python
a = area_circle(1)
```

at what's known as the "top level" or "main level" of execution and

   - Jumps back up to the function

   - Assigns 1 to the parameter `radius`.

   - Runs the code inside the function using `radius` as a variable inside the function.

- Returns the result of the calculation back to the top level and assigns the value 3.14159 to the variable `a`.

5. Repeats the process of running the function at the second `print()`, this time with the parameter value 2 and therefore a new value for `radius` inside the function.

6. Repeats the process of running the function right after we reassign `a`, this time with parameter value 75.1 taken from the variable `r`.

8.6 Flow of Control

- To re-iterate, the "flow of control" of Python here involves
  - Reading the function definition without executing the function.
  - Seeing a "call" to the function, jumping up to the start of the function and executing its code.
  - Returning the result of the function back to the place in the program that called the function and continuing the execution.
- Functions can compute many different things and return any data type Python supports.

8.7 Arguments, Parameters and Local Variables

- *Arguments* are the values 1, 2 and 75.1 in our above examples.
- These are each passed to the *parameter* called `radius` named in the function header. This parameter is used just like a variable in the function.
- The variable `pi` and `area` are *local variables* to the function (though we should probably use the `math` module for `pi` in the future).
- Neither `pi` nor `radius` nor `area` exists at the top / main level. At this level, they are ''undefined variables''. Try it out.

8.8 Exercise / Example

As some basic practice, we'll write a function to convert miles per hour to kilometers per day, and then we'll write several calls to demonstrate its use. Then we will give you time to work on the first lecture exercise.

8.9 A More Complicated Example

- We'll use the example below to illustrate two important concepts:
  1. Functions can call other functions, ones that we write ourselves or that Python provides.
  2. Functions may have multiple parameters. One argument in the function call is required for each parameter. Arguments and parameters are matched up *in order*.
     - There are ways to override this, but we will not study them yet.
- In the example we will switch to the use of `math.pi` and we will use this from now on.

8.10 Computing the Surface Area of A Cylinder Using Two Functions

- Here is the Python code, in file `lec05_surface_area.py`:

```python
import math

def area_circle(radius):
    return math.pi * radius ** 2

def area_cylinder(radius,height):
    circle_area = area_circle(radius)
    height_area = 2 * radius * math.pi * height
    return 2*circle_area + height_area

print('The area of a circle of radius 1 is', round(area_circle(1),2))
r = 2
height = 10
print('The surface area of a cylinder with radius', r)
print('and height', height, 'is', round(area_cylinder(r,height),2))
```

- Now we've defined two functions, one of which calls the other.

- Flow of control proceeds in two different ways here:

    1. Starting at the first `print()` function call at the top level, into `area_circle()` and back.

    2. At the third `print()`

        (a) into `area_cylinder()`

        (b) into `area_circle()`

        (c) back to `area_cylinder()`, and

        (d) back to the top level (and then into `round()` and finally into `print()`).

- The Python interpreter keeps track of where it is working and where to return to when it is done with a function, even if it is back into another function.

- What are the arguments, the parameters, the local variables and the global variables?

8.11 Practice Problems

1. Write a function that computes the area of a rectangle. Then, write a second function that calls this function three times to compute the surface area of a rectangular solid.

2. Write a function that returns the middle value among three integers. (Hint: make use of `min()` and `max()`). Write code to test this function with different inputs.

You will notice that the solution to the first problem in particular is longer than the solution without using functions. While we don't often write such short functions in practice, here it is a good illustration.

8.12 More on program structure

- Let us revisit the program structure that will allow us to write readable programs.

    – First a general comment describing the program.

      – Second, all import statements.

      – Third, all function definitions.

      – Fourth, the main body of your program.

- Well structured programs are easy to read and debug. We will work hard to help you develop good habits early on.

8.13 Thinking About What You See

Why is it NOT a mistake to use the same name, for example `radius`, in different functions (and sometimes at the top level)?

8.14 Let's Make Some Mistakes

In order to check our understanding, we will play around with the code and make some mistakes on purpose

- Removing `math` from `math.pi` in one definition
- Changing the name of a function
- Switching the order of the parameters in a function call
- Making an error in our calculation
- Calling `print()` with the result of a function that does not return a value

8.15 A Final Example, Including Documentation

- We will write a function that linearly scales a test score, so that if `raw` is the "raw score" then the scaled score will be

```
scaled = a*raw + b
```

The values of `raw`, `a` and `b` will be parameters of the function. We will also want to cap the scaled score at 100.

- In writing our function we will be careful to document the meaning of the parameters and the assumptions. You should get into the habit of doing this.

8.16 Why Functions?

Our goal in using functions is to write code that is

- Easier to think about and write
- Easier to test: we can check the correctness of `area_circle` before we test `area_cylinder`.
- Clearer for someone else to read
- Reusable in other programs

Together these define the notion of *encapsulation*, another important idea in computer science!

8.17 Summary

- Functions for encapsulation and reuse

- Function syntax

- Arguments, parameters and local variables

- Flow of control, including functions that call other functions

- You can find the code developed in this class under the class modules for Lecture 5.

8.18 Additional Practice:

The lecture exercises provide just the beginnings of what you need to do to practice with the concepts and with writing short problems. Here is some additional material for review before our next class. It goes all the way back to Lecture 2. . .

- **Expressions:** What type of data do they return?

- Try typing simple math formulas to the Python interpreter like:

```
>>> 1 + 2 * 3 / 3 * 4**2 **3 - 3 / 3*4
```

and manually find the output. Don't be fooled by the spaces! Operator precedence is in effect. Try writing your own expressions.

- **Variables:** Do you know what are valid and invalid variable names?

- What is the difference in the output between:

```
>>> 3 + 4
>>> print(3 + 4)
>>> x = 3 + 4
>>> print(x)
>>> print(x = 3+4)      # This causes an error...
```

Try to guess before typing them in, but make a habit of typing simple statements like this and looking at the result.

- **Assignment:** Can you trace the value of a variable after many different assignments? Don't be fooled by the name of variables. Try to do it manually:

```
>>> one = 2
>>> two = 1
>>> three = 4
>>> one += 3 * two
>>> two -= 3 * one + three
```

By the way, make a habit of picking nice variable names. Your variables should be meaningful whenever possible both to you and to anyone else reading your code.

- **Functions:** Write the functions from class on your own using the Python interpreter. Try to do it without looking at notes. Can you do it?

  - Write a function that returns a value.

  - Write a function with no return.

  - Write a function where `return` is not the last statement in the function.

– Call these functions by either printing their result or assigning their results to a value. Here, I'll get you started.

```python
def regenerate_doctor(doctor_number):
    return doctor_number+1

def regenerate_tardis(doctor_number):
    print "Tardis is now ready for doctor number", doctor_number

def eliminate_doctor(doctor_number):
    return 0
    print "You will be eliminated doctor", doctor_number
```

– Write functions that use the built-in functions. Make sure you memorize what they are and how they are used.

- Finally, write some functions to a file and call them from within the file. Now, execute the file. By next class, make sure all of this is quite easy to do without consulting the course notes.

LECTURE 5 — EXERCISES

Solutions to the problems below must be sent to Submitty for grading. A separate file must submitted for each problem. These must be submitted by 4 pm on Friday, February 2.

1. Write a function called `convert2fahren` that takes a Celsius temperature and converts it to Fahrenheit, returning the answer. Write code that calls the function three times to convert temperatures 0, 32, and 100 to Fahrenheit, printing the result each time. To keep things simple for these exercises, the output should be

```
0 -> 32.0
32 -> 89.6
100 -> 212.0
```

   Submitty will check that a function exists with exactly the given name, that it does the calculation, and that it is called three times.

2. Write a function called `frame_string` that takes a string as an argument. Its job is to print that string with a frame around it, just like in Lab 2. Unlike the other functions we have written `frame_string` does not need and therefore should not have a `return` statement. Write code to call the function two times. For the first call pass the string `Spanish Inquisition`. For the second call, pass the string `Ni`. Print a blank line between calls. The output should be

```
************************
** Spanish Inquisition **
************************

********
** Ni **
********
```

   In addition to checking the output, we will check that you wrote a function called `frame_string` and that your code calls this function twice.

For extra practice, but not to be uploaded to Submitty, solve the following:

1. Add code to your first function that has a second parameter which is the minimum temperature in Fahrenheit. If the converted temperature is less than this, use this minimum value. Write code to run this modified function using the value 459.67 and test that it works correctly.

2. Add code to the second program you wrote to print out the result of the calls to `frame_string`. The output should be the value `None`. This is a special Python value that indicates "there is no value". The most common example of using `None` is the result of a function that has no `return` statement.

TEN

LECTURE 6 — DECISIONS

10.1 Initial Example:

- Suppose we have a height measurements for two people, Chris and Sandy. We have the tools to write a program that determines which height measurement is greater:

```python
chris_height = float(input("Enter Chris's height (in cm): "))
sandy_height = float(input("Enter Sandy's height (in cm): "))
print("The greater height is", max(chris_height,sandy_height))
```

- But, we don't have the tools yet to *decide* who has the greater height. For this we need *if statements*:

```python
chris_height = float(input("Enter Chris's height (in cm): "))
sandy_height = float(input("Enter Sandy's height (in cm): "))
if chris_height > sandy_height:
    print("Chris is taller")
else:
    print("Sandy is taller")
```

- This is the first of many lectures exploring logic, if statements, and decision making.

10.2 Overview — Logic and Decision Making

- Boolean logic
- Use in decision making
- Use in branching and alternatives

Reading: Chapter 5 of *Practical Programming*. We will not cover all of this today, and will return to the rest of this chapter later in the semester.

10.3 Part 1: Boolean Values

- Yet another type
- Values are only `True` and `False`
- We'll see a series of operations that either produce or use boolean values, including relational operators such as <, <=, etc. and logical operations such as `and` and `or`.
- We can assign them to variables, as in,

```
x = True
```

although we will not explore this much during the current lecture.

10.4 Relational Operators — Less Than and Greater Than

- Comparisons between values, perhaps values associated with variables, to produce a boolean outcome.

- For numerical values, <, <=, >, >= are straightforward:

```
>>> x = 17
>>> y = 15.1
>>> x < y
False
>>> x <= y
False
>>> x <= 17
True
>>> y < x
True
```

- The comparison operators <, <=, >, >= may also be used for strings but the results are sometimes a bit surprising:

```
>>> s1 = 'art'
>>> s2 = 'Art'
>>> s3 = 'Music'
>>> s4 = 'music'
>>> s1 < s2
False
>>> s2 < s3
True
>>> s2 < s4
True
>>> s1 < s3
False
```

- With strings, the ordering is what's called *lexicographic* rather than purely alphabetical order:

    - All capital letters come before small letters, so strict alphabetical ordering can only be ensured when there is no mixing of caps and smalls.

10.5 Relational Operators: Equality and Inequality

- Testing if two values are equal uses the combined, double-equal symbol == rather than the single =, which is reserved for assignment.

    - Getting accustomed to this convention requires practice, and it is a common source of mistakes

- Inequality is indicated by !=.

- We will play with a few examples in class.

10.6 Part 1 Exercises

We will stop here and give students a chance to work on the first lecture exercise.

10.7 Part 2: if Statements

- General form of what we saw in the example we explored at the start of lecture:

```
if condition:
    block1
else:
    block2
```

  where

  - `condition` is the result of a logical expression, such as the result of computing the value of a relational operation
  - `block1` is Python code executed when the condition is `True`
  - `block2` is Python code executed when the condition is `False`
- All statements in the `block1` and `block2` must be indented the same number of spaces
- The `block` continues until the indentation stops, and returns to the same level of indentation as the statement starting with `if`
- The `else:` and `block2` are optional, as the following example shows.

10.8 Example: Heights of Siblings

- Here is a more extensive version of our initial example, implemented using two consecutive `if` statements and not using an `else`:

```
name1 = "Dale"
print("Enter the height of", name1, "in cm ==> ", end='')
height1 = int(input())

name2 = "Erin"
print("Enter the height of", name2, "in cm ==> ", end='')
height2 = int(input())

if height1 < height2:
    print(name2, "is taller")
    max_height = height2

if height1 >= height2:
    print(name1, "is taller")
    max_height = height1

print("The max height is", max_height)
```

- Writing two separate `if` statements like this, while good as an illustration, is not a good idea in practice. We need to read the code to understand that the two `if` statements produce mutually exclusive results. Instead we should use `else`

```
name1 = "Dale"
height1 = int(input("Enter the height of " + name1 + " in cm ==> "))

name2 = "Erin"
height2 = int(input("Enter the height of " + name2 + " in cm ==> "))

if height1 < height2:
    print(name2, "is taller")
    max_height = height2
else:
    print(name1, "is taller")
    max_height = height1

print("The max height is", max_height)
```

- Notes:

    - The blank lines are added for clarity; they are not required for these programs to have correct syntax.

    - Neither program handles the case of Dale and Erin being the same height. For this we need the next Python construct.

10.9 Elif

Recall the kids guessing game where someone thinks of a number and you have to guess it. The only information you are given is that the person who knows the number tells you if your guess is too high, too low, or if you got it correct.

- When we have three or more alternatives to consider we use the if-elif-else structure:

```
if condition1:
    block1
elif condition2:
    block2
else:
    block3
```

- We'll rewrite the height example to use `elif` to handle the case of Dale and Erin having the same height.

- Notes:

    - You do **NOT** need to have an `else` block.

    - Exactly **one** block of code (block1, block2, block3) is executed! Don't forget this!

    - If we leave off the `else:` and block3, then it is possible that none of the blocks are executed.

    - You can use multiple `elif` conditions and blocks.

10.10 Part 3: More Complex Boolean Expressions, Starting with and

Consider the following piece of Python code that outputs a message if it was above freezing both yesterday and today

```
cel_today = 12
cel_yesterday = -1
if cel_today > 0 and cel_yesterday > 0:
    print("It was above freezing both yesterday and today.")
```

- A boolean expression involving and is True if and only if **both** the relational operations produce the value True

10.11 More Complex Boolean Expressions — or

Consider the following:

```
cel_today = 12
cel_yesterday = -1
if cel_today > 0 or cel_yesterday > 0:
    print("It has been above freezing in the last two days.")
```

- The boolean expression is True if ANY of the following occurs

    - the left relational expression is True,

    - the right relational expression is True,

    - **both** the left and right relational expression are True.

- This is called the *inclusive or* and it is somewhat different from common use of the word *or* in English.

- For examples, in the sentence

    You may order the pancakes or the omelet.

usually means you may choose pancakes, or you may choose an omelet, but you may not choose both (unless you pay extra).

    - This is called the *exclusive-or*; it is only used in logic and computer science in very special cases.

    - Hence, or always means *inclusive-or*.

10.12 Boolean Logic — not

- We can also "logically negate" a boolean expression using not.

```
a = 15
b = 20
if not a<b:
    print("a is not less than b")
else:
    print("a is less than b")
```

10.13 Final Example - Is a Point Inside a Rectangle

We'll gather all of ideas from class to solve the following example: Suppose the bounds of a rectangle are defined by

```
x0 = 10
x1 = 16
y0 = 32
y1 = 45
```

A point at location x, y is inside the rectangle if

$$x0 < x < x1 \quad \text{and} \quad y0 < y < y1.$$

Under what conditions are we outside of the rectangle? Under what conditions are on the boundary?

We will write a program that reads in an x, y coordinate of a point and outputs a message depending on whether the point is inside the rectangle, outside the rectangle, or on the boundary. The final code will be posted on the course website.

10.14 Summary and Looking Ahead

- if-else and if-elif-else are tools for making decisions and creating alternative computations and results
- The conditional tests involve relationship operators and logical operators
    - Be careful of the distinction between = and ==
- In Lecture 11 we will review boolean logic and discuss more complex if structures.

LECTURE 6 — EXERCISES

Solutions to the problems below must be sent to Submitty for grading. A separate file must submitted for each problem. These must be submitted by 4 pm on Tuesday, February 6th.

1. Consider the following boolean expressions:

```
a = 1.6
b = -1.7
c = 15
s = 'hi'
t = "good"
u = "Bye"
v = "GOOD"
w = "Bye"
y = 15.1

a < b              # A
a < abs(b)         # B
a >= c             # C
s < t              # D
t == v             # E
u == w             # F
b < y              # G
```

Upload a text file to Submitty containing only the labels (A, B, C, etc.) of the lines that evaluate to `True`. Each line of the uploaded file should contain a single capital letter and the letters should be in alphabetical order.

2. Consider the following boolean expressions:

```
x = 15
y = -15
z = 32
x == y and y < z        #A
x == y or y < z         #B
x == abs(y) and y < z   #C
x == abs(y) or y < z    #D
not x == abs(y)         #E
not x != abs(y)         #F
```

Upload a text file to Submitty containing only the labels (A, B, C, etc.) of the lines that evaluate to `True`. Each line of the file should contain a single capital letter and the letters should be in alphabetical order.

3. So far we have assumed all input to our programs is correct. In practice, however, programs must do extensive error checking. Here is a slightly-contrived problem to illustrate this: Write a short program that asks the user to input two numbers where one of them must be greater than 10 and the other must be less than or equal to 10. It does not matter which is which. If both inputs are greater than 10, the program should

output the error message "Both are above 10." If both are less than or equal 10, the program should output the message "Both are below 10." If the one of the numbers is above 10 and the other is less than or equal to 10, no message should be output. Regardless of any messages, the program should then output the average of the two numbers, accurate to 2 decimals. This program must use one if, one elif and **no** else. Note: just like in HW 1, the program should output a value immediately after reading it. Also, if you are having problems matching our output format, explore the difference between the output of the following two lines

```
print("{:.2f}".format(112.099))
print(round(112.099, 2))
```

Here are two examples of how your program might look when run from the interpreter:

```
Enter the first number: 17.1
17.1
Enter the second number: 13.45
13.45
Both are above 10.
Average is 15.28
```

and

```
Enter the first number: 4.7
4.7
Enter the second number: 15.5
15.5
Average is 10.1
```

LECTURE 7 — TUPLES, MODULES, IMAGES

12.1 Overview

- While most of this lecture is not covered in our text book, this lecture serves as an introduction to using more complex data types like lists.

- We will first learn a simple data type called tuples which allow us to work with multiple values together - including returning two or more values from a function.

- We will then revisit modules, how functions you write can be used in other programs.

- Most of the class we will be learning how to use a new module for manipulating images.

  - We will introduce a new data type - an image - which is much more complex than the other data types we have learned so far.

  - We will study a module called *pillow* which is specifically designed for this data type.

- Class will end with a *review* for Monday's Exam 1, so it will be a bit long. . .

12.2 Tuple Data Type

- A *tuple* is a simple data type that puts together multiple values as a single unit.

- A Tuple allows you to access individual elements: first value starts at zero (this "indexing" will turn into a big Computer Science thing!)

```
>>> x = (4, 5, 10)    # note the parentheses
>>> print(x[0])
4
>>> print(x[2])
10
>>> len(x)
3
```

- As we will explore in class tuples and strings are similar in many ways.

```
>>> s = 'abc'
>>> s[0]
'a'
>>> s[1]
'b'
```

- Just like strings, you cannot change a part of the tuple; you can only change the entire tuple!

```
>>> x[1] = 2
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> s[1] = 'A'
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

12.3 What are tuples good for?

- Tuples are Python's way of making multiple assignments.

```
>>> 2,3
(2, 3)
>>> x = 2,3
>>> x
(2, 3)
>>> a,b=x
>>> a
2
>>> b
3
>>> c,d=3,4
>>> c
3
>>> d
4
```

- You can write functions that return multiple values.

```
def split(n):
    ''' Split a two-digit number into its tens and ones digit '''
    tens = n // 10
    ones = n % 10
    return (tens, ones)

x = 83
ten,one = split(x)
print( x, "has tens digit", ten, "and ones digit", one )
```

outputs

```
83 has tens digit 8 and ones digit 3
```

- We can do the reverse, passing a tuple to a function.

```
def combine( digits ):
    return digits[0]*100 + digits[1]*10 + digits[2]

d = (5, 2, 7)
print( combine(d))
```

outputs

```
527
```

12.4 First Lecture Exercise

We will take five minutes and work on the (only) two lecture exercises.

12.5 Basics of modules

- Recall that a module is a collection of Python variables, functions and objects, all stored in a file.

- Modules allow code to be shared across many different programs.

- Before we can use a module, we need to import it. The import of a module and use of functions within the module have the follow general form.

```
>>> import module_name
>>> module_name.function(arguments)
```

12.6 Area and Volume Module

- Here are a number of functions from the area calculations we've been developing so far, gathered in a single Python file called `lec08_area.py`:

```python
import math

def circle(radius):
    ''' Compute and return the area of a circle '''
    return math.pi * radius**2

def cylinder(radius,height):
    ''' Compute and return the surface area of a cylinder '''
    circle_area = circle(radius)
    height_area = 2 * radius * math.pi * height
    return 2*circle_area + height_area

def sphere(radius):
    '''  Compute and return the surface area of a sphere '''
    return 4 * math.pi * radius**2
```

- Now we can write another program that imports this code and uses it:

```python
import lec08_area

r = 6
h = 10
a1 = lec08_area.circle(r)        #  Call a module function
a2 = lec08_area.cylinder(r,h)    #  Call a module function
a3 = lec08_area.sphere(r)        #  Call a module function
print("Area circle {:.1f}".format(a1))
print("Surface area cylinder {:.1f}".format(a2))
print("Surface area sphere {:.1f}".format(a3))
```

- We will review this in class.

12.7 PIL/PILLOW — Python Image Library

- PILLOW is a series of modules built around the `Image` type, our first object type that is not part of the main Python language

    – We have to tell Python about this type through `import`

- We will use images as a continuing example of what can be done in programming beyond numbers and beyond text.

- See

    http://pillow.readthedocs.org/en/latest/handbook/tutorial.html

    for more details.

12.8 Images

- An image is a two-dimensional matrix of pixel values
- The origin is in the upper left corner, see below:
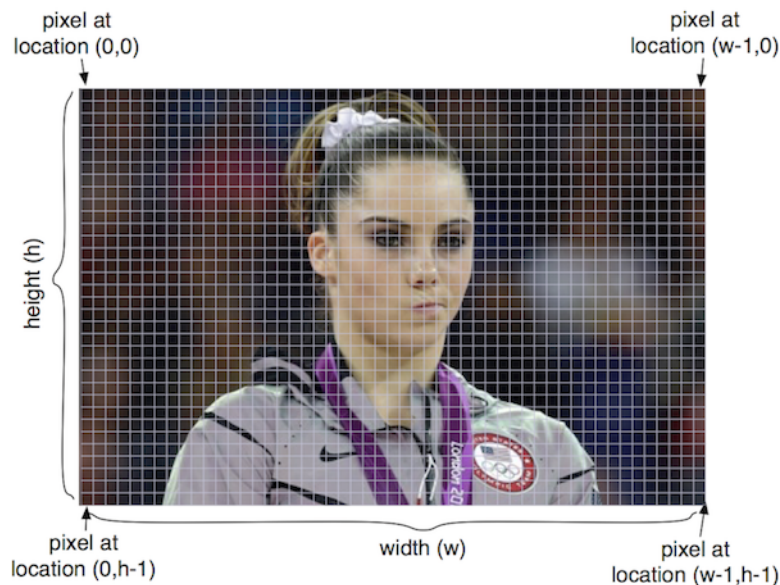
- Pixel values stored in an image can be:

    – RGB — a "three-tuple" consisting of the red, green and blue values, all non-negative integers

    – L — a single "gray-scale" integer value representing the brightness of each pixel

- Some basic colors:

Color	(red,green,blue) value
Black	(0,0,0)
Red	(255,0,0)
Green	(0,255,0)
Blue	(0,0,255)
White	(255,255,255)
Light Gray	(122,122,122)

12.9 Some important image modules

- **Image** module contains main functions to manipulate images: open, save, resize, crop, paste, create new images, change pixels, etc.

- **ImageDraw** module contains functions to touch up images by adding text to it, draw ellipses, rectangles, etc.

- **ImageFont** contains functions to create images of text for a specific font.

- We will only use the Image module in this lecture.

12.10 Our First Image Program

- We'll start by working through the following example which you can save as `lec08_images_init.py`:

```python
from PIL import Image

filename = "chipmunk.jpg"
im = Image.open(filename)
print('\n' '*******************')
print("Here's the information about", filename)
print(im.format, im.size, im.mode)

gray_im = im.convert('L')
scaled = gray_im.resize( (128,128) )
print("After converting to gray scale and resizing,")
print("the image information has changed to")
print(scaled.format, scaled.size, scaled.mode)

scaled.show()
scaled.save(filename + "_scaled.jpg")
```

12.11 Image Type and Methods

- Let us now see some very useful image methods. You need to be very careful with the image functions.

  - Some functions do change the image and return nothing.

  - Some functions do not change the image and return a value, which is sometimes a new image.

  It is crucial that you use each function correctly.

- `im = Image.open(filename)` **reads an image with the given filename** and returns an image object (which we are associating with the variable `im`).

– Because we only give the file name, and not a more complete path, the Python script and the image must be stored in the same folder.

- Images are complex objects. They have associated properties that you can print or use. For example

```
>>> im = Image.open('swarm.jpg')
>>> im.size
(600, 800)
>>> im.format
'JPEG'
>>> im.mode
'RGB'
```

You can see that `im.format` and `im.mode` are strings, while `im.size` is a tuple. All of these are values associated with an image object.

- `im.show()` is a function that displays the image.

- `im.save(filename)` saves the image in the given file name

- You can create an empty new image with given dimensions using: `Image.new("RGB",(width,height))`.

```
>>> im5 = Image.new('RGB', (200,200))
>>> im5.show()
```

- You can also create a new image by cropping a part of a given image:

```
>>> im.crop((w1,h1,w2,h2))
```

which will crop a box from upper left corner (w1, h1) to lower right corner (w2,h2).

You can see that the box is entered as a tuple.

The image object `im` is not changed by this function, but a new image is returned. So, we must assign it to a new variable.

Try this:

```
>>> im2 = im.crop((100,100,300,400))
>>> im2.show()
>>> im.show()
```

- You can get a new image that is a resized version of an existing image. The new size must be given as a tuple: `im.resize((width,height))`

```
>>> im3 = im.resize((300,200))
>>> im3.save('resized.jpg')
```

- `im.convert(mode)` creates a copy of in image with a new mode - gray scale (`'L'`) in the following example:

```
>>> im4 = im.convert('L')
>>> im4.show()
```

12.12 Something new, functions that change an image

- The functions we have seen so far return a new result, but never change the object that they apply to.

- More complex types such as images, often provide methods that allow us to change the object (image) for efficiency reason.

- You just have to remember how each function works.

- Here is our first function with this property: `im1.paste(im2, (x,y))` pastes one image (im2) into the first image (im1) starting at the top left coordinates (x,y). The first image is changed as a result, but not the second one.

  Note that the second image must fit in the first image starting with these coordinates; otherwise, the pasted image will be cropped.

- How we call such a function is different:

```
>>> im1 = Image.open('sheep.jpg')
>>> im1.size
(600, 396)
>>> im = Image.new('RGB', (600, 396*2))
>>> im.paste( im1, (0,0))    ##not assigning the result of paste to a new variable
>>> im.show()
>>> im.paste( im1, (0, 396))
>>> im.show()
```

- The fact that the function `paste` changes an image is an implementation decision made by the designers of PIL, mostly because images are so large and copying is therefore time consuming.

  Later in the semester, we will learn how to write such functions.

12.13 Example 2: Cut and pasting parts of an image

- This example crops three boxes from an image, creates a new image and pastes the boxes at different locations of this new image.

```
from PIL import Image

im = Image.open("lego_movie.jpg")
w,h = im.size

## Crop out three columns from the image
## Note: the crop function returns a new image
part1 = im.crop((0,0,w//3,h))
part2 = im.crop((w//3,0,2*w//3,h))
part3 = im.crop((2*w//3,0,w,h))

## Create a new image
newim = Image.new("RGB",(w,h))

## Paste the image in different order
## Note: the paste function changes the image it is applied to
newim.paste(part3, (0,0))
newim.paste(part1, (w//3,0))
newim.paste(part2, (2*w//3,0))
newim.show()
```

12.14 Summary

- Tuples are similar to strings and numbers in many ways. You cannot change a part of a tuple. However, unlike other simple data types, tuples allow access to the individual components using the indexing notation [ ].

- Modules contain a combination of functions, variables, object definitions, and other code, all designed for use in other Python programs and modules

- `PILLOW` provides a set of modules that define the `Image` object type and associated methods.

12.15 Reviewing for the exam: topics and ideas

Here are crucial topics to review before the exam.

- Syntax: can you find syntax errors in code?

- Correct variable names, assigning a value to a variable

- Output: can you predict the output of a piece of code?

- Expressions and operator precedence

- The distinction between integer and float division

- The distinction between division (4//5) and modulo (4%5) operators, and how they work for positive and negative numbers

- Remember shorthands: +=, -=, /=, *=.

- Functions: defining functions and using them

- Distinguish between variables local to functions and variables that are global

- Modules: how to import and call functions that are from a specific module (`math` is the only one we learned so far)

- How to access variable values defined in a module (see `math.pi` for example)

- Strings: how to create them, how to escape characters, multi-line strings

- How to use `input()`: remember it always returns a string

- Boolean data type: distinguish between expressions that return integer/float/string/Boolean

- Remember the distinction between = and ==

- Boolean value of conditions involving AND/OR/NOT

- `if/elif/else`: how to write them. Understand what parts are optional and how they work

- Remember the same function may work differently and do different things when applied to a different data type.

- Review all about the different ways to call the print function for multiple lines of input

- Operators: + (concatenation and addition), * (replication and multiplication), /, %, **

- Functions: `int()`, `float()`, `str()`, `math.sqrt()`, `min()`, `max()`, `abs()`, `round()`, `sum()` etc.

- Functions applied to string objects using the dot notation, where `string` is a string object, such as `"car"` or the name of a string variable:

  - `string.upper()`, `string.lower()`, `string.replace()`, `string.capitalize()`, `string.title()`, `string.find()`, `string.count()`, `len()`

- Distinguish between the different types of functions we have learned in this class:

  - Functions that take one or more values as input and return something (input objects/values are not modified)

```
>>> min(3,2,1)
1
>>> mystr = 'Monty Python'
>>> len(mystr)
12
```

- Functions that take one or more values as input and return nothing (input objects/values are not modified)

```
>>> def print_max(val1, val2):
...     print("Maximum value is", max(val1, val2))
...
>>> x1 = 10
>>> x2 = 15
>>> print_max(x1, x2)
Maximum value is 15
```

- Functions that apply to an object, like a string, and return a value (but do not modify the object that they are applied to)

```
>>> mystr = 'Monty Python'
>>> mystr.replace('o','x')
'Mxnty Pythxn'
>>> mystr
'Monty Python'
>>> mystr.upper()
'MONTY PYTHON'
>>> mystr
'Monty Python'
```

- Functions that are applied to an object, like an Image and modify it (but not return anything), we have only learned Image.paste so far (and images will NOT be on the exam).

```
>>> im.paste( im2, (0,0) )
```

- Local vs. global variables: Can you tell what each of the print statements print and explain why?

```
def f1(x,y):
    return x+y

def f2(x):
    return x+y

x = 5
y = 10

print('A:', f1(x,y))
print('B:', f1(y,x))
print('C:', f2(x))
print('D:', f2(y))
print('E:', f1(x))
```

12.16 Reviewing for the exam: problem solving

In the remaining time we will go through several practice questions to demonstrate how we approach these problems. While our immediate concern is the exam, you will be developing your problem solving skills and programming abilities. Most of these questions have appeared on previous exams in CS 1.

1. What is the **exact** output of the following Python code? What are the global variables, the function arguments, the local variables, and the parameters in the code?

```python
x=3

def do_something(x, y):
    z=x+y
    print(z)
    z += z
    print(z)
    z += z * z
    print(z)

do_something(1,1)
y=1
do_something(y,x)
```

2. Write a Python function that takes two strings as input and prints them together on one 35-character line, with the first string left-justified, the second string right-justified, and as many periods between the words as needed. For example, the function calls

```python
print_left_right( 'apple', 'banana')
print_left_right( 'syntax error', 'semantic error')
```

should output

```
apple.......................banana
syntax error.........semantic error
```

You may assume that the lengths of the two strings passed as arguments together are less than 35 characters.

3. In the United States, a car's fuel efficiency is measured in miles driven per gallon used. In the metric system it is liters used per 100 kilometers driven. Using the values 1.609 kilometers equals 1 mile and 1 gallon equals 3.785 liters, write a Python function that converts a fuel efficiency measure in miles per gallon to one in liters per 100 kilometers and returns the result.

4. Write a program that reads Erin's height (in cm), Erin's age (years), Dale's height (in cm) and Dale's age (years) and tells the name of the person who is both older and taller or tells that neither is both older and taller.

LECTURE 7 — EXERCISES

Solutions to the problems below must be sent to Submitty for grading. A separate file must be submitted for each problem. These must be submitted by 4 pm on Friday, February 9th.

1. What is the output from the following code. Note that we did not cover all of these techniques in class, so you might need to do some exploration on your own.

```python
def hmmm( x ):
    if x[0] > x[1]:
        return (x[1], x[0])
    else:
        return x

s = ('a', 'b')
t = (1, 2, 3)
u = (4, 5, 2)
print( t[1] + u[0] )
print( t+u )
print( s[1] * t[2] )
print( hmmm(u) )
print( hmmm( (5, 2, 3) ))
```

2. Write a function called `add_tuples` that takes three tuples, each with two values, and returns a single tuple with two values containing the sum of the values in the tuples. Test your function with the following calls:

```python
print(add_tuples( (1,4), (8,3), (14,0) ))
print(add_tuples( (3,2), (11,1), (-2,6) ))
```

Note that these two lines of code should be at the bottom of your program. This should output

```
(23, 7)
(12, 9)
```

LECTURE 8 — LISTS PART 1

14.1 Overview

- So far we've looked at working with individual values and variables.

- This is cumbersome even for just two or three variables.

- We need a way to aggregate multiple values and refer to them using a single variable.

- We have done a little bit of this with strings and tuples, but now we are going to get started for real.

This lecture is largely based on Sections 8.1-8.3 of *Practical Programming*.

14.2 Lists are Sequences of Values

- Gather together values that have common meaning.

- As a first example, here are scores of 7 judges for the free skating part of a figure skating competition:

```
scores = [ 59, 61, 63, 63, 68, 64, 58 ]
```

- As a second example, here are the names of the planets in the solar system (including Pluto, for now):

```
planets = [ 'Mercury', 'Venus', 'Earth', 'Mras', 'Jupiter',
    'Saturn', 'Neptune', 'Uranus', 'Pluto' ]
```

- Notes on syntax:

    - Begin with [and end with]

    - Commas separate the individual values

    - The spaces between values are optional and are used for clarity here.

    - Any type of object may be stored in a list, and each list can mix different types.

14.3 Why bother?

- Gather common values together, providing them with a common name, **especially** when we don't know how many values we will have.

- Apply an operation to the values as a group.

- Apply an operation to each value in the group.

- Examples of computations on lists:

    – Average and standard deviation

    – Which values are above and below the average

    – Correct mistakes

    – Remove values (Pluto)

    – Look at differences

- Watch for these themes throughout the next few lectures.

14.4 Accessing Individual Values — Indexing

- Notice that we made the mistake in typing `'Mras'`. How do we fix this? We'll start by looking at *indexing*.
- The line

```
>>> print(planets[1])
```

accesses and prints the string at what's known as index 1 of the list `planets`.

- Each item / value in the list is associated with a unique index
- Indexing in Python (and most other programming languages) starts at 0.
- The notation is again to use `[` and `]` with an integer (non-negative) to indicate which list item.
- What is the last index in `planets`?

    – We can find the length using `len()` and then figure out the answer.

14.5 A Memory Model for Lists

We'll draw a memory model in class that illustrates the relationship among

- The name of the list
- The indices
- The values stored in the list

14.6 Practice Problems

We will work on these in class:

1. What is the index of the first value in `scores` that is greater than 65?

2. Write a line of Python code to print this value and to print the previous and next values of the list.

3. What is the index of the middle value in a list that is odd length? For even length lists, what are the indices of the middle two values?

14.7 Changing Values in the List

- Once we know about indexing, changing a value in a list is easy:

```
>>> planets[3] = 'Mars'
```

- This makes item 3 of `planets` now refer to the string `'Mars'`

- Now we can check the output:

```
>>> print(planets)
```

to make sure we got it right.

- Strings are similar in many ways.

```
>>> s = 'abc'
>>> s[0]
'a'
>>> s[1]
'b'
```

- Big difference: you can change a part of a list; you cannot change part of a string!

```
>>> s[1] = 'A'
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

14.8 All Indices Are Not Allowed

- If `t` is a list, then the items are stored at indices from 0 to `len(t)-1`.

- If you try to access indices at `len(t)` or beyond, you get a run-time error. We'll take a look and see.

- If you access negative indices, interesting things happen:

```
>>> print(planets[-1])
Pluto
```

- More specifically, for any list `t`, if `i` is an index from 0 to `len(t)-1` then `t[i]` and `t[i-len(t)]` are the same spot in the list.

14.9 Functions on Lists: Computing the Average

- There are many functions (methods) on lists. We can learn all about them using the `help` command.

  - This is just like we did for strings and for modules, e.g.

```
>>> import math
>>> help(math)

>>> help(str)
```

- Interestingly, we can run help in two ways, one

```
help(list)
```

gives us the list methods, and the second

```
help(planets)
```

tells us that `planets` is a list before giving us list methods.

- First, let's see some basic functions on the list values.

- The basic functions `max`, `sum` and `min` may be applied to lists as well.

- This gives us a simple way to compute the average of our list of scores.

```
>>> print("Average Scores = {:.2f}".format( sum(scores) / len(scores) ))
Average Scores = 62.29
>>> print("Max Score =", max(scores))
Max Score = 68
>>> print("Min Score =", min(scores))
Min Score = 58
```

- Exploring, we will look at what happens when we apply `sum`, `max` and `min` to our list of planet names. Can you explain the result?

14.10 Functions that modify the input: Sorting a list

- We can also sort the values in a list by sorting it. Let's try the following:

```
>>> planets = [ 'Mercury', 'Venus', 'Earth', 'Mras', 'Jupiter', \
'Saturn', 'Neptune', 'Uranus', 'Pluto' ]
>>> planets
['Mercury', 'Venus', 'Earth', 'Mras', 'Jupiter', 'Saturn', 'Neptune', 'Uranus', 'Pluto']
>>> planets.sort()
>>> planets
['Earth', 'Jupiter', 'Mercury', 'Mras', 'Neptune', 'Pluto', 'Saturn', 'Uranus', 'Venus']
```

- Note that we did not assign the value returned by sort to a new variable. This is the first function we have learned outside of our Image module that modifies the input but returns nothing. Try the following and see what happens:

```
>>> scores = [ 59, 61, 63, 63, 68, 64, 58 ]
>>> new_scores = scores.sort()
>>> scores
[58, 59, 61, 63, 63, 64, 68]
>>> new_scores
>>>
```

  - Ok, what is the value of the variable `new_scores`? It is unclear. Let's try in a different way.

```
>>> print(scores)
[58, 59, 61, 63, 63, 64, 68]
>>> print(new_scores)
None
>>>
```

- So, the function returns nothing! But, it does change the value of the input list.

- It does so because lists are containers, and functions can manipulate what is inside containers. Functions cannot do so for simple types like integer and float.

- If we want a new list that is sorted without changing the original list then we use the `sorted()` function:

```
>>> scores = [ 59, 61, 63, 63, 68, 64, 58 ]
>>> new_scores = sorted(scores)
>>> scores
[ 59, 61, 63, 63, 68, 64, 58 ]
>>> new_scores
[58, 59, 61, 63, 63, 64, 68]
>>>
```

14.11 More Functions: Appending Values, Inserting Values, Deleting

- Now, we will see more functions that can change the value of a list without returning anything.

- Armed with this knowledge, we can figure out how to add and remove values from a list:

  - `append()`

  - `insert()`

  - `pop()`

  - `remove()`

- These operations are fundamental to any "container" — an object type that stores other objects.

  - Lists are our first example of a container

14.12 Lists of Lists

- Note that lists can contain any mixture of values, including other lists.

- For example, in

```
>>> L = [ 'Alice', 3.75, ['MATH', 'CSCI', 'PSYC' ], 'PA' ]
```

  - `L[0]` is the name,

  - `L[1]` is the GPA

  - `L[2]` is a list of courses

  - `L[2][0]` is the 0th course, `'MATH'`

  - `L[3]` is a home state abbreviation

- We will write code to print the courses, to change the math course to a stats course, and to append a zipcode.

14.13 Additional Practice Problems

1. Write three different ways of removing the last value — `'Pluto'` — from the list of planets. Two of these will use the method `pop`.

2. Write code to insert `'Asteroid belt'` between `'Mars'` and `'Jupiter'`.

14.14 Summary

- Lists are sequences of values, allowing these values to be collected and processed together.

- Individual values can be accessed and changed through indexing.

- Functions and methods can be used to **return** important properties of lists like `min()`, `max()`, `sum()`.

- Functions and methods can be also used to **modify** lists, but not return anything.

LECTURE 8 — EXERCISES

Solutions to the problems below must be sent to Submitty for grading. A separate file must submitted for each problem. These must be submitted by 4 pm on Friday, February 16.

1. Upload a text file showing the output of the following code to Submitty. As usual you should try to predict the output by hand.

```python
l1 = [ 6, 12, 13, 'hello' ]
print(l1[1], l1[-2])
l1.append( 15 )
print( len(l1) )
print( len(l1[3]) )
l1.pop(3)
l1.sort()
l1.insert(2, [14, 15] )
l1[3]  += l1[4]
l1[3]  += l1[2][1]
print(l1[3])
l1.pop()
l1[2].remove(14)
print(l1)
```

2. Write a short Python program that starts with the list

```python
values = [ 14, 10, 8, 19, 7, 13 ]
```

(The above statement list should be the first line of your program.) Then add code that does the following steps:

(a) Reads an integer, prints it (as we have done for input when using Submitty), and appends it to the end of `values`.

(b) Reads another integer, prints it and inserts it at index location 2 of `values`.

(c) Prints the integer at index 3 of `values` and print the integer at index -1 of `values`, both on one line.

(d) Prints the difference between the maximum and minimum of the integers in `values`.

(e) Prints the average of the integers in `values`, accurate to one decimal place. This must use the functions `sum()` and `len()`.

(f) Prints the median of the numbers in `values`. Since the list is even length (a fact that you are allowed to use, just for this exercise), this is the average of the two middle integers after `values` is sorted.

Here is an example of running our solution (as it would look on Submitty):

```
Enter a value: 15
Enter another value: 23
8 15
Difference: 16
Average: 13.6
Median: 13.5
```

LECTURE 9 — WHILE LOOPS

16.1 Overview

- Loops are used to access and modify information stored in lists and are used to repeat computations many times.
- We construct loops using logical conditions: `while` loops
- We will investiage single and multiple loops

Reading: Our coverage of loops is in a different order than that of *Practical Programming*. A direct reference for reading material is Section 9.6.

16.2 Part 1: The Basics

- Loops allow us to repeat a block of code multiple times. This is the basis for many sophisticated programming tasks
- We will see two ways to write loops: using `while` loops and `for` loops
- In Python, `while` loops are more general in Python because you can always write a `for` loop using a `while` loop.
- We will start with `while` loops first and then see how we can simplify common tasks with `for` loops later.

16.3 Basics of While

- Our first `while` loop just counts numbers from 1 to 9, and prints them.

```
i=1
while i<10:
    print(i)
    i += 1    ## if you forget this, your program will never end
```

- General form of a `while` loop:

```
while condition:
    block
```

- Steps

    1. Evaluate any code before `while`

2. Evaluate the `while` loop `condition`:

    (a) If it is `True`, evaluate the block of code, and then repeat the evaluation of the condition.

    (b) If it is `False`, end the loop, and continue with the code after the loop.

In other words, the cycle of evaluating the condition followed by evaluating the block of code continues until the condition evaluates to `False`.

- An important issue that is sometimes easy to answer and sometimes very hard to answer is to know that your loop will always terminate.

16.4 Using Loops with Lists

- Often, we use loops to repeat a specific operation on every element of a list.

- We must be careful to create a number that will serve as the index of elements of a list. Valid values are: 0 up to (not including) the length of list.

```
co2_levels = [ (2001, 320.03), (2003, 322.16), (2004, 328.07),\
               (2006, 323.91), (2008, 341.47), (2009, 348.92),\
               (2010, 357.29), (2011, 363.77), (2012, 361.51),\
               (2013, 382.47) ]

i=0
while i< len(co2_levels):
    print( "Year", co2_levels[i][0], \
           "CO2 levels:", co2_levels[i][1])
    i += 1
```

- Let's make some errors to see what happens to the loop.

16.5 Part 1 Practice

1. Write a while loop to count down from 10 to 1, printing the values of the loop counting variable `i` (it could be some other variable name as well).

2. Modify your loop to print the values in the list `co2_levels` in reverse order.

16.6 Accumulation of values

- Often, we use loops to accumulate some type of information such as adding all the values in a list.

- Let's change the loop to add numbers from 1 to 9.

```
i=1
total = 0
while i<10:
    total += i
    i += 1
print(total)
```

(Of course you can and should do this with the `sum()` function, but guess what that actually does!)

- Let's use a loop to add the numbers in a list.

      – Now, we will use the numbers the loop generates to index a list.

      – So, the loop must generate numbers from 0 to length of the list.

```python
co2_levels = [ (2001, 320.03), (2003, 322.16), (2004, 328.07),\
               (2006, 323.91), (2008, 341.47), (2009, 348.92),\
               (2010, 357.29), (2011, 363.77), (2012, 361.51),\
               (2013, 382.47) ]

i=0
total=0
while i< len(co2_levels):
    total += co2_levels[i][1]
    i += 1

print("Total co2_levels is", total)
```

- Let's have a more interesting example. Be very careful not to use an incorrect index for the list.

      – Count the number of CO2 values in the list that are greater than 350.

      – Calculate and print the percentage change in each measurement year from the previous measurement year.

      – Determine the years in which the CO2 levels dropped compared to the previous measurement. Output just these years.

16.7 Part 2 Practice

1. Suppose we wanted to print the first, third, fifth, etc. elements in a list. Write code to accomplish this.

```python
months=['jan','feb','mar','apr','may','jun','jul','aug','sep','oct','nov','dec']
```

2. Now, use a similar loop code to print a little evergreen tree.

```
    *
   ***
  *****
 *******
*********
   ***
   ***
```

3. Try this later: change your loop to work for any size evergreen.

16.8 Loops that end on other conditions

- Here is a while loop to add the non-zero numbers that the user types in.

```python
total = 0
end_found = False

while not end_found:
    x = int( input("Enter an integer to add (0 to end) ==> "))
    if x == 0:
```



```
            end_found = True
        else:
            total += x

print(total)
```

- We will work through this loop by hand in class.

16.9 Multiple nested loops

- Loops and if statements can both be nested.

- We've already seen this for if statements.

- Here's an example where we print every pair of values in a list.

- First solution:

```
L = [2, 21, 12, 8, 5, 31]
i = 0
while i < len(L):

    j = 0
    while j < len(L):
        print(L[i], L[j])
        j += 1

    i += 1
```

- This solution prints the values from the same pair of indices twice - e.g.the pair 21, 12 is printed once when *i=1, j=2* and once when *i=2, j=1*.

- How can we modify it so that we only print each pair once?

- What has to change if we don't want to print when *i==j*?

- Finally, we will modify the resulting loop to find the two closest values in the list.

16.10 Summary

- Loops are especially useful for iterating over a list, accessing its contents, and adding or counting the values from a list. This is done in the `sum()` and `len()` functions of Python.

- Each loop has a stopping condition — the boolean expression in the *while* statement. The loop will end when the condition evaluates to *True*.

- If the stopping condition is never reached, the loop will become "infinite".

- Often a counter variable is used. It (a) is given an initial value before the loop starts, (b) is incremented (or decremented) once in each loop iteration, and (c) is used to stop the loop when it reaches the index of the end (or beginning) of the list.

- We will demonstrate a simple way to write these common loops with a `for` loop in the next lecture.

LECTURE 9 — EXERCISES

Solutions to the problems below must be sent to Submitty for grading. A separate file must submitted for each problem. Each problem requires you to use a *while* loop. As a whole this exercise set is a bit longer than previous ones because loops are so important and require so much practice. Solutions must be submitted by 4 pm on Wednesday, February 21.

1. Write a program that asks the user for a single integer `n` and prints the non-negative multiples of 3 that are less than `n`. Here is an example run of the program on Submitty

```
Enter a positive integer: 12
0
3
6
9
```

2. The following list represents the population of New York State (in hundreds of thousands of people) for the US Census in 1790, 1800, 1810, etc., all the way through 2010.

```
census = [ 340, 589, 959, 1372, 1918, 2428, 3097, 3880, 4382, 5082, \
           5997, 7268, 9113, 10385, 12588, 13479, 14830, 16782, \
           8236, 17558, 17990, 18976, 19378 ]
```

Copy and paste this list into the start of a new program file. Then write code to find the average percentage change from one decade to the next, across all decades. For example, the change from 1790 to 1800 is (589 - 340) / 340 * 100 = 73.2% and the change from 1800 to 1810 is (959 - 589) / 589 * 100 = 62.8% so the average across just these two decades is 68.0%. The output of your program would then simply be

```
Average = 68.0%
```

Your answer will be different because it is taken from all decades.

3. Write a program that inputs integer values that the user types until the user types a 0. Each value (other than 0) should be stored in a list. The program should then output the minimum, maximum and average of the values in the list. Your program must start by creating an empty list to store the values. Here's an example of how it might look on Submitty:

```
Enter a value (0 to end): 5
Enter a value (0 to end): 3
Enter a value (0 to end): 11
Enter a value (0 to end): 0
Min: 3
Max: 11
Avg: 6.3
```

LECTURE 10 — LISTS PART 2

18.1 Topics

We will cover the first three of these topics during lecture. The fourth is for your own use:

- List aliasing, lists and functions
- For loops to operate on lists
- Slicing to create copies of lists and to create sublists
- Converting back and forth between strings and lists

18.2 List Aliasing

- Consider the following example Python code:

```
>>> L1 = [ 'RPI', 'WPI', 'MIT' ]
>>> L2 = L1
>>> L3 = [ 'RPI', 'WPI', 'MIT' ]
>>> L2.append( 'RIT' )
>>> L2[1] = 'CalTech'
>>> L1
['RPI', 'CalTech', 'MIT', 'RIT']
>>> L2
['RPI', 'CalTech', 'MIT', 'RIT']
>>> L3
['RPI', 'WPI', 'MIT']
```

- Surprised? This is caused by the creation of what we call an *alias* in computer science:
  - L1 and L2 reference the same list - they are *aliases* of each other and the underlying list - so changes made using either name change the underlying list
  - L3 references a different list that just happens to have the same string values in the same order: there would have been no confusion if the strings in the list had been different.
  - We'll use our memory model for lists to understand what is happening here.
- Python uses aliases for reasons of efficiency: lists can be quite long and are frequently changed, so copying of entire lists is expensive
- This is true for other *container* data types as well.
  - Assignments create an alias for images, lists, tuples, strings and, as we will see later, sets and dictionaries

> * Aliases of strings and tuples do not create the same confusion as other containers because they can not be changed once they are created.

- Fortunately, if we truly want to copy a list, Python provides a `copy()` method for lists. Try the following and see what happens.

```python
L1 = [1,2,3]
L2 = L1.copy()
L1.pop()
L2.append( 4 )
print(L1)
print(L2)
```

18.3 Aliasing and Function Parameters

- When a variable is passed to functions, a copy of its value is created if the value is a number or a booleans:

```python
def add_two(val1, val2):
    val1 += val2
    return val1

val1 = 10
val2 = 15
print(val1, val2)
print(add_two(val1,val2))
print(val1, val2)
```

- When a list is passed to functions, the parameter becomes an alias for the argument in the function call.

- Here is an example of a function that returns a list containing the two smallest values in its input list:

```python
def smallest_two(mylist):
    mylist.sort()
    newlist = []
    if len(mylist) > 0:
        newlist.append(mylist[0])
        if len(mylist) > 1:
            newlist.append(mylist[1])
    return newlist

values = [35, 34, 20, 40, 60, 30]

print("Before function:", values)
print("Result of function:", smallest_two(values))
print("After function:", values)
```

- In class we will discuss what happened.

18.4 What Operations Change a List? What Operations Create New Lists?

- Operations that change lists include

    - `sort`, `insert`, `append`, `pop`, `remove`

- Operations that create new lists
    - Slicing (discussed below), `copy()`, concatenation (+), replication (*) and `list()`

18.5 Part 1 Practice

Students will be given about 5 minutes to work on the first two lecture exercises

18.6 Part 2: For Loops and Operations on List Items

- Although *while* loops allow us to apply an operation to each entry in a list, Python has a construct called a *for* loop that is often easier to use for such operations.

- Our driving example will be the problem of capitalizing a list of names. We'll start with a simple example:

```python
animals = ['cat', 'monkey', 'hawk', 'tiger', 'parrot']
cap_animals = []
for animal in animals:
    cap_animals.append( animal.capitalize() )
print(cap_animals)
```

- We can understand what is happening by looking at this piece-by-piece:
    - The keyword `for` signals the start of a loop
    - `animal` is a loop variable that takes on the value of each item in the list (as indicated by the keyword `in`) in succession
        * This is called *iterating* over the values/elements of the list
    - The `:` signals the start of a block of code that is the "body of the loop", executed once in succession for each value that `animal` is assigned
    - The body of the loop here is just a single, indented line of code, but in other cases - just as using while loops - there may be more than one line of code.
    - The end of the loop body is indicated by returning to the same level of the indentation as the `for ...` line that started the loop.

18.7 Changing the Values in a List

- What if we wanted to change the list? We might consider copying `cap_animals` back to `animals` at the end of the code sequence.

- But this does not work if we wanted a function that capitalized all strings in a list.

```python
def capitalize_list( names ):
    cap_names = []
    for n in names:
        cap_names.append( n.capitalize() )
    names = cap_names

animals = ['cat', 'monkey', 'hawk', 'tiger', 'parrot']
capitalize_list(animals)
print(animals)      # Make sure you understand the output!!!
```

- This does not work because `names` is an alias for the list rather than the list itself!

- The following does not work either because `n` points to a string in the list (and when string values are changed, a new string is generated, effectively making `n` a copy of the value):

```
def capitalize_list( names ):
    for n in names:
        n = n.capitalize()
```

- So, based on what we know so far to actually change the values in the list we need to use indexing together with a `while` loop:

```
def capitalize_list( names ):
    i = 0
    while i < len(names):
        names[i] = names[i].capitalize()
        i += 1
```

- We can also solve this using a `for` loop and indexing, but for this we need a `range`

18.8 Using `range`

- A `range` "generates" values in a sequence, almost-but-not-quite like a list.

```
for i in range(5):
    print(i)
```

prints the values 0 through 4...

- We can convert a range to an actual list:

```
>>> x = list(range(5))
>>> print(x)
[0, 1, 2, 3, 4]
```

- The general form is

```
range( bi, ei, ii )
```

where

- `bi` is the initial value (defaults to 0)

- `ei` is the ending value (never included in the range!)

- `ii` is the increment, added each time (defaults to 1)

- We'll look at number of examples:

```
list(range(3,10))
list(range(4, 20, 4))
list(range(10, 2, -2))
```

- Using `for` loops on lists, we often use `len()` in combination with `range` to specify the indices that should be used.

```
def capitalize_list( names ):
    for i in range(len(names)):
        names[i] = names[i].capitalize()
```

Now we have our `for` loop based solution to capitalizing the names in a list.

- Unlike with a `while` loop there is no need to write code to compare our index / counter variable `i` directly against the bound and no need to write code to increment `i`.

- This use of `range` to generate an index list is common

  - When we want to change the integer, float or string values of a list.

  - When we want to work with multiple lists at once.

18.9 Part 2 Practice

1. Recall our list

```
co2_levels = [ 320.03, 322.16, 328.07, 333.91, 341.47, \
  348.92, 357.29, 363.77, 371.51, 382.47, 392.95 ]
```

For the purpose of this exercise only, please pretend the Python `sum` function does not exist, and then write a short section of Python code that uses a `for` loop to first compute and then print the sum of the values in the `co2_levels` list. You do not need to use indexing.

18.10 Using Indices to "Slice" a List and Create a New List

- Recall

```
co2_levels = [ 320.03, 322.16, 328.07, 333.91, 341.47, \
  348.92, 357.29, 363.77, 371.51, 382.47, 392.95 ]
```

- Now suppose we just want the values at indices 2, 3 and 4 of this in a new list:

```
>>> three_values = co2_levels[2:5]
>>> three_values
[328.07, 333.91, 341.47]
>>> co2_levels
[ 320.03, 322.16, 328.07, 333.91, 341.47, 348.92, 357.29, 363.77,
  371.51, 382.47, 392.95 ]
```

- We give the first index and one more than the last index we want

- If we leave off the first index, 0 is assumed, and if we leave off the second index, the length of the list is assumed.

- Negative indices are allowed — they are just converted to their associated positive values. Some examples:

```
>>> L1 = ['cat', 'dog', 'hawk', 'tiger', 'parrot']
>>> L1
['cat', 'dog', 'hawk', 'tiger', 'parrot']
>>> L1[1:-1]
['dog', 'hawk', 'tiger']
>>> L1[1:-2]
```

```
['dog', 'hawk']
>>> L1[1:-4]
[]
>>> L1[1:0]
[]
>>> L1[1:10]
['dog', 'hawk', 'tiger', 'parrot']
```

18.11 More on List Slicing

- Specifying indices for slicing and for a `range` are very similar:
    - A `range` uses () and is a generator, while slicing using [ ] and is applied to a list to create a new list.
- The most general form of slicing involves three values

```
L[si:ei:inc]
```

where

- `L` is the list

- `si` is the start index

- `ei` is the end index

- `inc` is the increment value

Any of the three values is optional

- We'll work through some examples in class to

- Use slicing to copy an entire list

- Use negative increments and generate a reversed list

- Extracting the even indexed values

- Note: `L[:]` returns a copy of the whole list of L. This is the same using method `L.copy()` or the function `list()`

```
>>> L2 = L1[:]
>>> L2[1] = 'monkey'
>>> L1
['cat', 'dog', 'hawk', 'tiger', 'parrot']
>>> L2
['cat', 'monkey', 'hawk', 'tiger', 'parrot']
>>> L3 = list(L1)
>>> L3[1] = 'turtle'
>>> L1
['cat', 'dog', 'hawk', 'tiger', 'parrot']
>>> L2
['cat', 'monkey', 'hawk', 'tiger', 'parrot']
>>> L3
['cat', 'turtle', 'hawk', 'tiger', 'parrot']
```

18.12 Concatenation and Replication

- Just like with strings, concatenation and replication can be applied to lists:

```
>>> v = [1,2,3]+[4,5]
>>> v
[1,2,3,4,5]
```

- and

```
>>> [1]*3
[1,1,1]
```

18.13 Part 3 Practice

1. What is the output of the following?

```
x = [6,5,4,3,2,1] + [7]*2
y = x
x[1] = y[2]
y[2] = x[3]
x[0] = x[1]
print(x)

y.sort()
print(x)
print(y)
```

2. Write a slicing command to extract values indexed by 1, 4, 7, 10, etc from a list L0 .

18.14 Converting Strings to Lists

- Version 1: use the function `list` to create a list of the characters in the string:

```
>>> s = "Hello world"
>>> t = list(s)
>>> print(t)
['H', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']
```

- Version 2: use the string `split` function, which breaks a string up into a list of strings based on the character provided as the argument.
    - The default is ' ':
    - Other common splitting characters are ',', '|' and '\t'
- We will play with the s = "Hello world" example in class.

```
>>> s.split()
['Hello', 'world']
>>> s = "Hello    worl  d"
>>> s.split()
['Hello', 'worl', 'd']
```

```
>>> s.split(' ')
['Hello', '', '', '', '', 'worl', '', '', 'd']
>>> s.split('l')
['He', '', 'o     wor', '   d']
```

18.15 Converting Lists to Strings

- What happens when we type the following?

```
>>> s = "Hello world"
>>> t = list(s)
>>> s1 = str(t)
```

  This is will not concatenate all the strings in the list (assuming they are strings).

- We can write a `for` loop to do this, but Python provides something simpler that works:

```
>>> L1 = [ 'No', 'one', 'expects', 'the', 'Spanish', 'Inquisition' ]
>>> print(''.join(L1))
NooneexpectstheSpanishInquisition
>>> print(' '.join(L1))
No one expects the Spanish Inquisition
```

  Can you infer from this the role of the string that the `join` funciton is applied to?

18.16 Indexing and Slicing Strings

- We can index strings:

```
>>> s = "Hello, world!"
>>> print(s[5])
,
>>> print(s[-1])
!
```

- We can apply all of the slicing operations to strings to create new strings:

```
>>> s = "Hello, world!"
>>> s[:len(s):2]
'Hlo ol!'
```

- Unlike lists, however, we can not use indexing to replace individual characters in strings:

```
>>> s[4] = 'c'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

18.17 Part 4 Practice

1. Given a list

```
L = [ 'cat', 'dog', 'tiger', 'lion' ]
```

Rewrite L so that it is a list of lists, with household pets in the 0th (sub)list, zoo animals in the first. Using slicing of L to create this new list and assign L to the result.

2. How can you append an additional list of farm animals (e.g. 'horse', 'pig' and 'cow') to L?

3. Write code to remove 'tiger' from the sublist of zoo animals.

4. Suppose you have the string

```
>>> s = "cat |  dog  | mouse | rat"
```

and you'd like to have the list of strings

```
>>> L = [ "cat", "dog", "mouse", "rat"]
```

Splitting the list alone does not solve the problem. Instead, you need to use a combination of splitting, and a loop that strips off the extra space characters from each string and appends to the final result. Write this code. It should be at most 4-5 lines of Python.

18.18 Summary

- Assignment of lists and passing of lists as parameters creates aliases of lists rather than copies.

- We use `for` loops to iterate through a list to work on each enty in the list.

- We need to combine `for` loops with indices generated by a `range` in order to change the contents of a list of integers, floats or strings. These indices are also used to work with multiple lists at once.

- Concatentation, replication and slicing create new lists.

- Most other list functions that modify a list do so without creating a new list: insert, sort, append, pop, etc.

- Strings may be indexed and sliced, but indexing may not be used to change a string.

- Conversion of a string to a list is accomplished using either `list` or `split`; conversion of a list of strings to a string uses `join`.

18.19 Additional Review Exercises: What Does Python Output?

1. Without typing into the Python interpreter, find the outputs from the following operations:

```
>>> x = ['a','b','c','d', 'e']
>>> print(x)

>>> for item in x:
...     print("{}".format(item))
...

>>> print(x[3])

>>> x[3] = 3
>>> x

>>> len(x)
```

```
>>> x[2]=x[1]
>>> x

>>> x[5]

>>> y = x[1:4]

>>> y

>>> x
```

2. What about these operations?

```
>>> y = [1, 2, 3]

>>> y.append('cat')

>>> y

>>> y.pop()

>>> y


>>> y.remove(2)
>>> y

>>> y.remove('cat')

>>> z = ['cat','dog']
>>> z.insert(1,'pig')
>>> z.insert(0,'ant')
>>> z

>>> z.sort()
>>> z

>>> z1 = z[1:3]
>>> z1

>>> z
```

3. Write a function that returns a list containing the smallest and largest values in the list that is passed to it as an argument *without changing the list*? Can you think of several ways to do this?

    (a) Using `min` and `max`

    (b) Using sorting (but remember, you can't change the original list)

    (c) Using a `for` loop that searches for the smallest and largest values.

LECTURE 10 — EXERCISES

Solutions to the problems below must be sent to Submitty for grading. A separate file must submitted for each problem. Due date is 4 pm on Tuesday, October 27th.

1. Submit a text file showing the output of the following code:

```python
L1 = [ 1, 5, [5, 2], 'hello' ]
L2 = L1
L3 = L1.copy()
L2.append( 4 )
L1.append( 3 )
print(L1)
print(L2)
print(L3)
```

2. Submit a text file showing the output of the following code

```python
def head_and_tail(L):
    from_back = L.pop()
    from_front = L.pop(0)
    L.append(from_front)
    L.insert(0,from_back)

L1 = [ [1, 2], 3 ]
L3 = L1.copy()
L2 = L1
L2[-1] = 5
L2.insert(1,6)
head_and_tail(L1)

print(L1[0], L1[-1], len(L1))
print(L2[0], L2[-1], len(L2))
print(L3[0], L3[-1], len(L3))
```

3. The solution to this problem and the two that follow should start with the assignment

```python
co2_levels = [ 320.03, 322.16, 328.07, 333.91, 341.47, \
               348.92, 357.29, 363.77, 371.51, 382.47, 392.95 ]
```

Write a Python program that prints the number of values that are greater than the average of the list. For this you may use Python's `sum` and `len` functions and you must use a `for` loop. Do NOT use a `range`, however, and do not use indexing.

Your output should simply be:

```
Average: 351.14
Num above average: 5
```

4. Suppose we discovered that the measurement of CO2 values was uniformly too low by a small fraction p. Write a function that increases each value in `co2_levels` by the fraction p. (In other words if x is the value before the increase then x*(1+p) is the value after.) For this problem you need to use a `range`, `len()` and indexing. Start by asking the user for the percentage. Output the first and last values of the revised list. Your program should end with the lines

```
print('First: {:.2f}'.format(co2_levels[0]))
print('Last: {:.2f}'.format(co2_levels[-1]))
```

Here is an example of running your program:

```
Enter the fraction: 0.03
First: 329.63
Last: 404.74
```

5. Write a function called `is_increasing` that returns `True` if the values in the list it is passed are in increasing order and `False` otherwise. Use a `for` loop and indexing to accomplish this. Test the function with the following main code:

```
print('co2_levels is increasing: {}'.format(is_increasing(co2_levels)))
test_L1 = [ 15, 12, 19, 27, 45 ]
print('test_L1 is increasing: {}'.format(is_increasing(test_L1)))
test_L2 = [ 'arc', 'circle', 'diameter', 'radius', 'volume', 'area' ]
print('test_L2 is increasing: {}'.format(is_increasing(test_L2)))
test_L3 = [ 11, 21, 19, 27, 28, 23, 31, 45 ]
print('test_L3 is increasing: {}'.format(is_increasing(test_L3)))
```

These should be the only `print()` function calls in the code you submit.

LECTURE 11 — DECISIONS PART 2

20.1 Overview — Logic and Decision Making, part 2

- Program structure

- Debugging and "hardening" a simple function

- A bit of design and a bit of a random walk

- More on logic - the key to getting programs right

    - Boolean logic

    - Nested if statements

    - Assigning boolean variables

20.2 Part 1: Program Structure

Programming requires four basic skills:

1. Develop a solution.

    - Start with small steps, solve them, and then add complexity

2. Structure your code.

    - Move repeated code into functions

    - Make your code easy to read and modify. Use meaningful variable names and break complex operations into smaller parts.

    - Place values in variables so they are easy to change.

    - Include comments for important functions, but do not clutter your code with unnecessary comments. A classic example of a completely unnecessary comment is

    ```
    x += 1    # increment x
    ```

    - Document assumptions about what is passed to a function.

    - If your function is meant to return a value, make sure it always does.

3. Test your code.

    - Find all possible cases that your program should handle, including typos in the input. As programs get larger, this is increasingly hard.

- If you cannot check for all inputs, then you must then check your code for meaningful inputs: regular (expected inputs) and edge cases (inputs that can break your code).
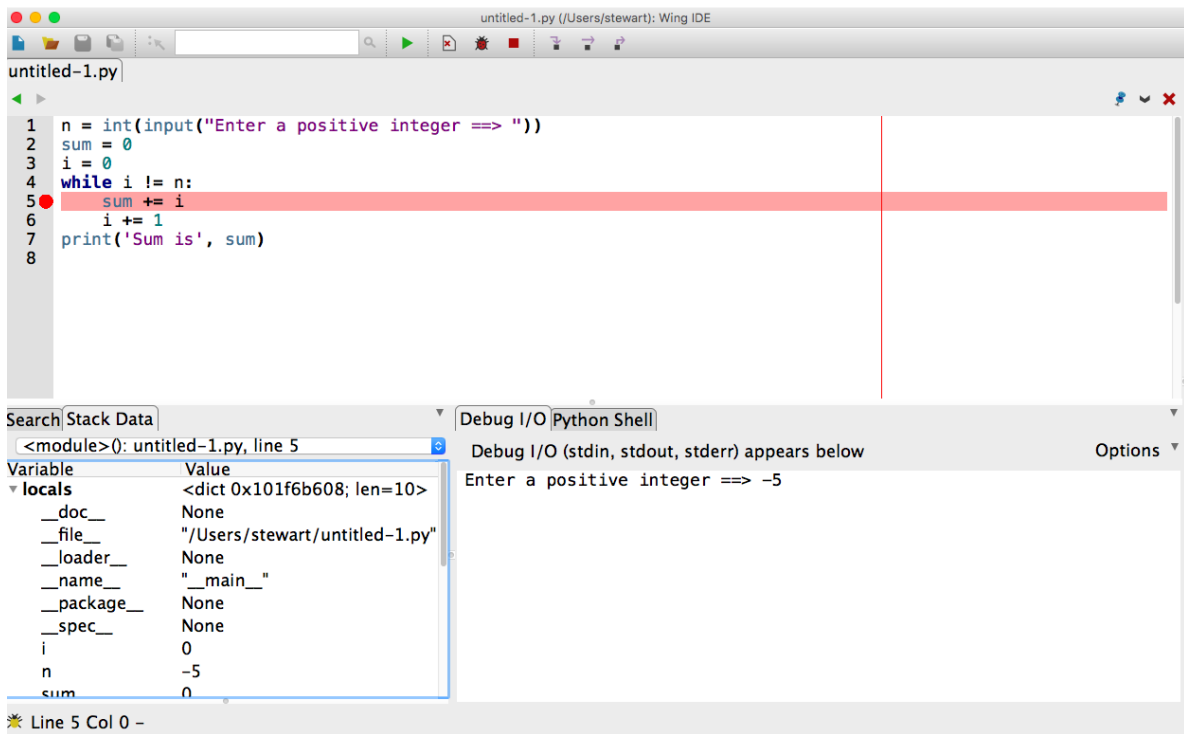
4. Debug your code.

- Never assume an untested part of your code is bug free. "If it ain't tested, it's broken."

- Learn syntax well so that you can spot and fix syntax errors fast.

- Semantic errors are much harder to find and fix. You need strategies to isolate where the error is.

- Print output before and after crucial steps.

- Look at where your program crashed.

- Fix the first error first, not the biggest error. The first error may be the cause of bigger errors later in your code.

- Use a debugger.

- Simplify the problem: remove (by commenting out) parts of your code until you no longer have an error. Look at the last code removed for a source of at least part of your errors.

- Test parts of your code separately and once you are convinced they are bug free, concentrate on other parts.

20.3 Help with debugging

- Consider the following code to add the first `n` integers:

```python
n = int(input("Enter a positive integer ==> "))
total = 0
i = 0
while i != n:
    total += i
    i += 1
print('Sum is', total)
```

- Does it work? For all inputs? Might it run forever? (We'll ignore the fact that a `for` loop would be better here.)

- How might we find such an error?

  – Careful reading of the code

  – Insert print statements

  – Use the Wing IDE debugger.

- We will practice with the Wing IDE debugger in class, using it to understand the behavior of the program. We will explain the following picture

---

and note the use of

- The hand, bug and stop symbols on the top of the display, and

- The Debug I/O and Stack Data at the bottom of the display.

- Debugging becomes crucial in tracking logic errors as well.

20.4 Program organization

- Envision your code as having two main parts: the main body and the functions that help the main code.

- Make sure your functions accomplish one well-defined task. This makes them both easy to test and useful in many places.

- As we will see in an example below, in Python it is good practice to separate the functions and the main body with the following addition to the program structure:

```
if __name__ == "__main__"
    # Put the main body of the program below this line
```

- This will have no apparent effect when a program is run. However, if a program is imported as a module into another program (like the utility code we have been giving you), any code within the above `if` block is skipped!

- This allows programs to work both as modules and stand alone code.

- When the primary purpose of your code is to provide functionality as a module, it is best to use the code in the main body to test the module functions.

20.5 Part 2: Extended Example of a Random Walk

- Many numerical simulations, including many video games, involve random events.

- Python includes a module to generate numbers at random. For example,

```python
import random

# Print three numbers randomly generated between 0 and 1.
print(random.random())
print(random.random())
print(random.random())

# Print a random integer in the range 0..5
print(random.randint(0,5))
print(random.randint(0,5))
print(random.randint(0,5))
```

- We'd like to use this to simulate a "random walk":

  - Hypothetically, a person takes a step forward or backward, completely at random (equally likely to go either way). This can be repeated over and over again until some stopping point is reached.

  - Suppose the person is on a platform with N steps and the person starts in the middle, this random forward/backward stepping process is repeated until they fall off (reach step 0 or step $N+1$)

    * "forward" is represented by an increasing step, while "backward" is represented by a decreasing step

  - How many steps does it take to fall off?

- Many variations on this problem appear in physical simulations.

- We can simulate a step in several ways:

  1. If `random.random()` returns a value less than 0.5 step backward; otherwise step forward.

  2. If `random.randint(0,1)` returns 1 then step forward; otherwise, step backward.

  3. Eliminate the `if` entirely and just increment by whatever `random.choice([-1,1])` returns (it will return either -1 (step backward) or 1 (step forward)).

- So, in summary, we want to start a random walk at position $N/2$ and repeatedly take a step forward or backward based on the output of the random number generator until the walker falls off.

- We will solve this problem together during lecture. We we start by enumerating some of the needed steps and then solving them individually before putting together the whole program.

  - Once we see the result we can think of several ways to change things and explore new questions and ideas. Remember, a program is never done!

20.6 Part 3: Review of Boolean Logic

- Invented / discovered by George Boole in the 1840's to reduce classical logic to an algebra

  - This was a crucial mathematical step on the road to computation and computers

- Values (in Python) are `True` and `False`

- Operators

– Comparisons: `<`, `>`, `<=`, `>=`, `==` `!=`

– Logic: `and`, `or`, `not`

20.7 Truth Tables

- Aside to recall the syntax: `and, or, not` are lower case!

- If we have two boolean expressions, which we will refer to as `ex1` and `ex2`, and if we combine their "truth" values using `and` we have the following "truth table" to describe the result

ex1	ex2	ex1 and ex2
False	False	False
False	True	False
True	False	False
True	True	True

- If we combine the two expressions using `or`, we have

ex1	ex2	ex1 or ex2
False	False	False
False	True	True
True	False	True
True	True	True

- Finally, using `not` we have

ex1	not ex1
False	True
True	False

20.8 DeMorgan's Laws Relating and, or, not

- Using `ex1` and `ex2` once again to represent boolean expressions, we have

```
not (ex1 and ex2) == (not ex1) or (not ex2)
```

- And,

```
not (ex1 or ex2) == (not ex1) and (not ex2)
```

- Also, distribution laws

```
ex1 and (ex2 or ex3) == (ex1 and ex2) or (ex1 and ex3)
ex1 or (ex2 and ex3) == (ex1 or ex2) and (ex1 or ex3)
```

- We can prove these using truth tables.

20.9 Why Do We Care?

- When we've written logical expressions into our programs, it no longer matters what we intended; it matters what the logic actually does.

- For complicated boolean expressions, we may need to almost prove that they are correct

20.10 Part 4: Additional Techniques in Logic and Decision Making

We will examine:

- Short-ciruiting

- Nested conditionals

- Storing the result of boolean expressions in variables

and then apply them to several problems

20.11 Short-Circuited Boolean Expressions

- Python only evaluates expressions as far as needed to make a decision.

- Therefore, in a boolean expression of the form

```
ex1 and ex2
```

ex2 will not be evaluated if ex1 evaluates to `False`. Think about why.

- Also, in a boolean expression of the form

```
ex1 or ex2
```

ex2 will not be evaluated if ex1 evalues to `True`.

- This "short-circuiting" is common across many programming languages.

20.12 Nested If Statements

- We can place `if` statements inside of other `if` statements.

- To illustrate, consider the following where `ex1`, `ex2`, `ex3` and `ex4` are all boolean expressions, and `blockA`, `blockB`, `blockD` and `blockE` are sections of code.

```
if ex1:
    if ex2:
        blockA
    elif ex3:
        blockB
elif ex4:
    blockD
else:
    blockE
```

- We will examine this example in class and answer the following questions:
    - Under what conditions is each block executed?
    - Is it possible that no blocks are executed?
    - What is the equivalent non-nested if-elif-else structure?

20.13 Storing the Result of a Boolean Expression

- Sometimes we store the result of boolean expressions in a variable for later use:

```
f = float(input("Enter a Fahrenheit temperature: "))
is_below_freezing = f < 32.0
if is_below_freezing:
    print("Brrr.  It is cold")
```

- We use this to
    - Make code clearer
    - Avoid repeated evaluation of the same expression, especially if the expression requires a lot of computation.

20.14 Examples for Lecture

We will work on the following examples during class, as time permits.

1. In the following code, for what values of x and y does the code print 1, for what values does the code print 2, and for what values does the code print nothing at all?

```
if x>5 and y<10:
    if x<5 or y>2:
        if y>3 or z<3:
            print 1
    else:
        print 2
```

The moral of the story is that you should be careful to ensure that your logic and if structures **cover the entire range of possibilities**!

2. Doctors sometimes assess a patient's risk of heart disease in terms of a combination of the BMI (body mask index) and age using the following table:

	Age ≤ 45	Age > 45
BMI < 22.0	Low	Medium
BMI ≥ 22.0	Medium	High

Assuming the values for a patient are stored in variables age and bmi, we can write the following code

```
slim = bmi < 22.0
young = age <= 45
```

We will work out two different ways of printing *Low, Medium* or *High* according to the table based on the values of the boolean variables slim and young.

---

3. Challenge example: Suppose two rectangles are determined by their corner points - (x0,y0) and (x1,y1) for one rectangle and (u0,v0) and (u1,v1) for the other. Write a function that takes these four tuples as arguments and returns `True` when the two rectangles intersect and `False` otherwise.

20.15 Summary of Discussion of If Statements and Logic

- Logic is a crucial component of every program.

- Basic rules of logic, including DeMorgan's laws, help us to write and understand boolean expressions.

- It sometimes requires careful, precise thinking, even at the level of a proof, to ensure logical expressions and if statement structures are correct.

  - Many bugs in supposedly-working programs are caused by conditions that the programmers did not fully consider.

- If statements can be structured in many ways, sometimes nested several levels deep.

  - Nesting deeply can lead to confusing code, however.

  - Warning specifically for Python: you can easily change the meaning of your program by accidentally changing indentation. It is very hard to debug these changes.

- Using variables to store boolean values can make code easier to understand and avoids repeated tests.

- Make sure your logic and resulting expressions cover the universe of possibilities!

LECTURE 11 — EXERCISES

Solutions to the problems below must be sent to Submitty for grading. A separate file must submitted for each problem. Solutions must be submitted by 4 pm on Friday February 23.

1. Suppose you have a tuple that stores the semester and year a course was taken, as in

```python
when = ('Spring',2015)
```

Write a function called `earlier_semester()` that takes two such tuples as arguments and returns `True` if the first tuple represents an earlier semester and `False` otherwise. The possible semesters are `'Spring'` and `'Fall'`. (Just to be clear, Fall 2013 is later than Spring 2013.) Test it using the following main code (you can cut-and-paste from the browser), which should be your only print function calls.

```python
# Insert your function def here...

w1 = ('Spring',2015)
w2 = ('Spring',2014)
w3 = ('Fall', 2014)
w4 = ('Fall', 2015)
print( "{} earlier than {}? {}".format( w1, w2, earlier_semester(w1,w2)))
print( "{} earlier than {}? {}".format( w1, w1, earlier_semester(w1,w1)))
print( "{} earlier than {}? {}".format( w1, w4, earlier_semester(w1,w4)))
print( "{} earlier than {}? {}".format( w4, w1, earlier_semester(w4,w1)))
print( "{} earlier than {}? {}".format( w3, w4, earlier_semester(w3,w4)))
print( "{} earlier than {}? {}".format( w1, w3, earlier_semester(w1,w3)))
```

2. Suppose three siblings, `Dale`, `Erin` and `Sam`, have heights `hd`, `he` and `hs`, respectively. Write a program that reads in integer values of their heights and outputs the ordering from tallest to shortest. (For simplicity of this exercises, you may assume that the three heights you will be given are all different.) Try writing your solution using nested if statements and trying writing it without nested if statements; either is acceptable on Submitty and trying both is good practice. Here is an example as the code would look on Submitty.

```
Enter Dale's height: 124
Enter Erin's height: 112
Enter Sam's height: 119
Dale
Sam
Erin
```

LECTURE 12 — CONTROLLING LOOPS

22.1 Restatement of the Basics

- `for` loops tend to have a fixed number of iterations computed at the start of the loop

- `while` loops tend to have an indefinite termination, determined by the conditions of the data

- Most Python `for` loops are easily rewritten as `while` loops, but not vice-versa.

  - In other programming languages, `for` and `while` are almost interchangeable, at least in principle.

22.2 Overview of Today

- Ranges and control of loop iterations

- Nested loops

- Lists of lists

- Contolling loops through `break` and `continue`

Reading: *Practical Programming*, rest of Chapter 9.

22.3 Part 1: Ranges and For Loops— A Review

- A range is a function to generate a sequence of integers:

```
for i in range(10):
    print(i)
```

  outputs the digits 0 through 9 in succession, one per line.

  - Remember that this is up to and **not including** the end value specified!

- A range is not quite a list — instead it generates values for each successive iteration of a `for` loop.

  - For now we will convert each range to a list as the basis for studying them.

- If we want to start with something other than 0, we provide two integer values

```
>>> list(range(3,8))
[3, 4, 5, 6, 7]
```

- With a third integer values we can create increments. For example,

```
>>> list(range(4,20,3))
[4, 7, 10, 13, 16, 19]
```

starts at 4, increments by 3, stops when 20 is reached or surpassed.

- We can create backwards increments

```
>>> list(range(-1, -10, -1))
[-1, -2, -3, -4, -5, -6, -7, -8, -9]
```

22.4 Using Ranges in For Loops

- We can use the `range` to generate the sequence of loop variable values in a for loop. Our first example is printing the contents of the `planets` list

```
planets = [ 'Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter',
    'Saturn', 'Uranus', 'Neptune', 'Pluto' ]
for i in range(len(planets)):
    print(planets[i])
```

(In this case we don't need a index variable - we can just iterate over the values in the list.)

- The variable `i` is variously known as the *index* or the *loop* index variable or the subscript.

- We will modify the loop in class to do the following:

    - Print the indices of the planets (starting at 1!)

    - Print the planets backward.

    - Print every other planet.

22.5 Loops That Do Not Iterate Over All Indices

- Sometimes the loop index should not go over the entire range of indices, and we need to think about where to stop it *early*, as the next example shows.

- Example: Returning to our example from Lecture 1, we will briefly re-examine our solution to the following problem: Given a string, how can we write a function that decides if it has three consecutive double letters?

```
def has_three_doubles(s):
    for i in range(0, len(s)-5):
        if s[i] == s[i+1] and s[i+2] == s[i+3] and s[i+4] == s[i+5]:
            return True
    return False
```

- We have to think carefully about where to start our looping and where to stop!

- Refer back to Lecture 10 for further examples

22.6 Part 1 Practice

We will only go over a few of these in class, but you should be sure you can handle all of them

1. Generate a range for the positive integers less than 100. Use this to calculate the sum of these values, with and without (i.e. use `sum`) a for loop.

2. Use a range and a for loop to print the positive, even numbers less than the integer value associated with `n`.

3. Suppose we want a list of the squares of the digits 0..9. The following does NOT work

```
squares = list(range(10))
for s in squares:
    s = s*s
```

   Why not? Write a different for loop that uses indexing into the `squares` list to accomplish our goal.

4. The following code for finding out if a word has two consecutive double letters is wrong. Why? When, specifically, does it fail?

```
def has_two_doubles(s):
    for i in range(0, len(s)-5):
        if s[i] == s[i+1] and s[i+2] == s[i+3]:
            return True
    return False
```

22.7 Part 2: Nested Loops

- Some problems require *iterating* over either

    - two dimensions of data, or

    - all pairs of values from a list

- As an example, here is code to print all of the products of digits:

```
digits = range(10)
for i in digits:
    for j in digits:
        print("{} x {} = {}".format(i,j,i*j))
```

- How does this work?

    - for each value of `i` the variable in the first, or "outer", loop, Python executes the *entire* second, or `inner`, loop

    - Importantly, `i` stays fixed during the entire inner loop.

- We will look at finding the two closest points in a list.

22.8 Example: Finding the Two Closest Points

- Suppose we are given a list of point locations in two dimensions, where each point is a tuple. For example,

```
points = [ (1,5), (13.5, 9), (10, 5), (8, 2), (16,3) ]
```

- Our problem is to find the two points that are closest to each other.

    - We started working on a slightly simpler version of this problem at the end of Lecture 10.

- The natural idea is to compute the distance between any two points and find the minimum.

– We can do this with and without using a list of distances.

• Let's work through the approach to this and post the result on the course website.

22.9 Part 3: Lists of Lists

• In programming you often must deal with data much more complicated than a single list. For example, we might have a list of lists, where each list might be temperature (or pH) measurements at one location of a study site:

```
temps_at_sites = [ [ 12.12, 13.25, 11.17, 10.4],
                   [ 22.1, 29.3, 25.3, 20.2, 26.4, 24.3 ],
                   [ 18.3, 17.9, 24.3, 27.2, 21.7, 22.2 ],
                   [ 12.4, 12.5, 12.14, 14.4, 15.2 ] ]
```

• Here is code to find the site with the maximum average temperature; note that no indices are used.

```
averages = []
for site in temps_at_sites:
    avg = sum(site) / len(site)
    averages.append(avg)

max_avg = max(averages)
max_index = averages.index(max_avg)
print("Maximum average of {:.2f} occurs at site {}".format(max_avg, max_index))
```

• Notes:

– for loop variable `site` is an **alias** for each successive list in `temps_at_sites`

– A separate list is created to store the computed averages

– We will see in class how this would be written without the separate `averages` list.

22.10 Part 4: Controlling Execution of Loops

• We can control loops through use of

– `break`

– `continue`

• We need to be careful to avoid infinite loops

22.11 Using a Break

• We can terminate a loop immediately upon seeing the 0 using Python's `break`:

```
sum = 0
while True:
    x = int(input("Enter an integer to add (0 to end) ==> "))
    if x == 0:
        break
    sum += x
```

(continues on next page)

```
print(sum)
```

- `break` sends the flow of control immediately to the first line of code outside the current loop, and

- The while condition of `True` essentially means that the only way to stop the loop is when the condition that triggers the `break` is met.

22.12 Continue: Skipping the Rest of a Loop Iteration

- Suppose we want to skip over negative entries in a list. We can do this by telling Python to `continue` when it the loop variable, taken from the list, is negative:

```
for item in mylist:
    if item < 0:
        continue
    print(item)
```

- When it sees `continue`, Python immediate goes back to the "top" of the loop, skipping the rest of the code, and initiates the next iteration of the loop with a new value for `item`.

- Any loop that uses `break` or `continue` can be rewritten without either of these.

  - Therefore, we choose to use them only if they make our code clearer.

  - A loop with more than one `continue` or `break` is rarely well-structured, so if you find that you have written such a loop you should stop and rewrite your code.

- The example above, while illustrative, is probably better without the `continue`.

  - Usually when we use `continue` the rest of the loop is relative long. The condition that triggers the `continue` is tested at or near the top of the loop.

22.13 Termination of a While Loop

- When working with a while loop one always needs to ensure that the loop will terminate! Otherwise we have an *infinite loop*.

- Sometimes it is easy to decide if a loop will terminate. Sometimes it is not.

- Do either of the following examples cause an infinite loop?

```
import math
x = float(input("Enter a positive number -> "))
while x > 1:
    x = math.sqrt(x)
    print(x, flush=True)
```

```
import math
x = float(input("Enter a positive number -> "))
while x >= 1:
    x = math.sqrt(x)
    print(x, flush=True)
```

22.14 Summary

- `range` is used to generate a sequence of indices in a `for` loop.

- Nested for loops may be needed to iterate over two dimensions of data.

- Lists of lists may be used to specify more complex data. We process these using a combination of `for` loops, which may need to be nested, and Python's built-in functions. Use of Python's built-in functions, as illustrated in the example here in these notes, is often preferred.

- Loops (either for or while) may be controlled using `continue` to skip the rest of a loop iteration and using `break` to terminate the loop altogether. These should be used sparingly!

LECTURE 12 — EXERCISES

Solutions to the problems below must be sent to Submitty for grading. A separate file must submitted for each problem. Solutions must be submitted by 4 pm on Friday, March 2.

1. The following simple exercise will help you understand loops better. Show the output of each of the following pairs of `for` loops. The first two pairs are nested loops, and the third pair is formed by consecutive, or *sequential*, loops. Submit a single text file containing three lines, each with an integer on it.

```
# Version 1
sum = 0
for i in range(10):
    for j in range(10):
        sum += 1
print(sum)
```

```
# Version 2
sum = 0
for i in range(10):
    for j in range(i+1,10):
        sum += 1
print(sum)
```

```
# Version 3
sum = 0
for i in range(10):
    sum += 1
for j in range(10):
    sum += 1
print(sum)
```

2. Write a function called `first_day_greater` that takes two lists, L1 and L2, representing the daily measured weights of rat 1 and rat 2, respectively, and returns the index of the first day for which the weight for the first rat is greater than the weight of the second rat. If there are no such days then the function should return -1. You may NOT assume that L1 and L2 are the same length.

Use the following to test your program:

```
if __name__ == "__main__":
    L1 = [ 15.1, 17.3, 12.3, 16.4 ]
    L2 = [ 15.0, 17.7, 12.5, 16.9 ]
    print("Test 1: {}".format( first_day_greater(L1,L2) ))
    L2 = [ 15.6, 17.9, 18.2, 16.5, 12.7 ]
    print("Test 2: {}".format( first_day_greater(L1,L2) ))
    L2 = [ 15.9, 18.8, 11.4 ]
    print("Test 3: {}".format( first_day_greater(L1,L2) ))
```

3. Write a function called `find_min` that takes a list of lists and returns the minimum value across all lists. Test it with the following

```
if __name__ == "__main__":
    v = [ [ 11,12,3], [6, 8, 4], [ 17, 2, 18, 14] ]
    print("Min of list v: {}".format(find_min(v)) )
    u = [ [ 'car', 'tailor', 'ball' ], ['dress'], ['can', 'cheese', 'ring' ], \
              [ 'rain', 'snow', 'sun' ] ]
    print("Min of list u: {}".format(find_min(u)) )
```

LECTURE 13 — DATA FROM FILES AND WEB PAGES

24.1 Overview

- Review of string operations
- Files on your computer
    - Opening and reading files
    - Closing
    - Writing files
- Accessing files across the web
- Parsing basics
- Parsing html

Our discussion is only loosely tied to Chapter 8 of the text.

24.2 Review — String operations often used in file parsing

Let's review and go over some very common string operations that are particularly useful in parsing files.

- Remove characters from the beginning, end or both sides of a string with `lstrip`, `rstrip` and `strip`:

```
>>> x = "red! Let's go red! Go red! Go red!"
>>> x.strip("red!")
" Let's go red! Go red! Go "
>>> x.strip("edr!")
" Let's go red! Go red! Go "
>>> x.lstrip("red!")
" Let's go red! Go red! Go red!"
>>> x.rstrip("red!")
"red! Let's go red! Go red! Go "
>>> "    Go red!      ".strip()
'Go red!'
>>> " \n   Go red!    \t ".strip()
'Go red!'
>>> " \n   Go red!    \t ".strip(' ')
'\n   Go red!    \t'
```

With no arguments, the `strip` functions remove all white space characters.

- Split a string using a delimiter, and get a list of strings. Whitespace is the default delimiter:

```
>>> x = "Let's go red! Let's go red! Go red! Go red!"
>>> x.split()
["Let's", 'go', 'red!', "Let's", 'go', 'red!', 'Go', 'red!', 'Go', 'red!']
>>> x.split("!")
["Let's go red", " Let's go red", ' Go red', ' Go red', '']
>>> x.split("red!")
["Let's go ", " Let's go ", ' Go ', ' Go ', '']
>>> "Let's go red! \n Let's go    red! Go red! \t Go red!".split(" ")
["Let's", 'go', 'red!', '\n', "Let's", 'go', '', '', '', 'red!', 'Go', 'red!', '\t', 'Go',
↪'red!']
>>> "Let's go red! \n Let's go    red! Go red! \t Go red!".split()
["Let's", 'go', 'red!', "Let's", 'go', 'red!', 'Go', 'red!', 'Go', 'red!']
```

`split` returns the strings before and after the delimiter string in a list.

- The `find` function returns the first location of a substring in a string, and returns -1 if the substring is not found. You can also optionally give a starting and end point to search from:

```
>>> x
"Let's go red! Let's go red! Go red! Go red!"
>>> x.find('red')
9
>>> x.find('Red')
-1
>>> x.find("edr!")
-1
>>> x.find('red',10)
23
>>> x.find('red',10,12)
-1
>>> 'red' in x
True
>>> 'Red' in x
False
```

24.3 Opening and Reading Files

- On to our main topic....

- Given the name of a file as a string, we can open it to read:

```
f = open('abc.txt')
```

This is the same as

```
f = open('abc.txt','r')
```

- – Variable f now "points" to the first line of file abc.txt.

- – The 'r' tells Python we will be reading from this file — this is the default.

- One way to access contents of a file is by doing so one input line at a time. In particular,

```
line = f.readline()
```

reads in the next line up to and including the end-of-line character, and "advances" f to point to the next line of file abc.txt.

- By contrast,

```
s = f.read()
```

reads the entire **remainder** of the input file as a single string,

- storing the one (big) string in `s`, and
- advancing `f` to the end of the file!

- When you are at the end of a file, `f.read()` and `f.readline()` will both return the empty string: `""`.

24.4 Reading the contents of a file

- The most common way to read a file is illustrated in the following example that reads each line from a file and prints it on the screen:

```
f = open('abc.txt')
for line in f:
    print(line)
```

(Of course you can replace the call to `print()` with any other processing code since each `line` is just a string!)

- You can combine the above steps into a single for loop:

```
for line in open('abc.txt'):
    print(line)
```

24.5 Closing and Reopening Files

- The code below opens, reads, closes and reopens a file

```
f = open('abc.txt')

# Insert whatever code is need to read from the file
# and use its contents ...

f.close()
f = open('abc.txt')
```

- `f` now points again to the beginning of the file.
- This can be used to read the same file multiple times.

24.6 Example: Computing the Average Score

- We are given a file that contains a sequence of integers representing test scores, one score per line. We need to write a program that computes the average of these scores.
- Here is one solution

```
file_name = input("Enter the name of the scores file: ")
file_name = file_name.strip()    # Elminate extra white space that the user may have typed
print(file_name)

num_scores = 0
sum_scores = 0
for s in open(file_name):
    sum_scores += int(s)
    num_scores += 1

print("Average score is {:.1f}".format( sum_scores / num_scores ))
```

- In class we will discuss:

    - Getting the file name from the user

    - The importance of using strip

    - The rest of the program.

24.7 Writing to a File

- In order to write to a file we must first open it and associate it with a file variable, e.g.

```
f_out = open("outfile.txt","w")
```

- The "w" signifies *write mode* which causes Python to completely delete the previous contents of outfile.txt (if the file previously existed).

- It is also possible to use *append mode*:

```
f_out = open("outfile.txt","a")
```

which means that the contents of outfile.txt are kept and new output is added to the end of the file.

- Write mode is much more common than append mode.

- To actually write to a file, we use the write method:

```
f_out.write("Hello world!\n")
```

    - Each call to write passes only a **single string**.

    - Unlike what happens when using print, spacing and newline characters are required explicitly

    - You can use the format method of each string before you print it.

- You must close the files you write! Otherwise, the changes you made will not be recorded!

```
f_out.close()
```

24.8 Lecture Exercise 1:

You will have a few minutes to work on the first lecture exercise.

24.9 Opening Static Web Pages

- We can use the `urllib` module to access web pages.

- We did this with our very first "real" example:

```python
import urllib.request
word_url = 'http://www.greenteapress.com/thinkpython/code/words.txt'
word_file = urllib.request.urlopen(word_url)
```

- Once we have `words_file` we can use the `read`, `readline`, and `close` methods just like we did with "ordinary" files.

- When the web page is dynamic, we usually need to work through a separate API (application program interface) to access the contents of the web site.

24.10 Parsing

- Before writing code to read a data file or to read the contents of a web page, we must know the format of the data in the file.

- The work of reading a data file or a web page is referred to as *parsing*.

- Files can be of a fixed well-known format

  - Python code

  - C++ code

  - HTML (HyperText Markup Language, used in all web pages)

  - JSON (Javascript Object Notation, a common data exchange format)

  - RDF (resource description framework)

- Often there is a parser module for these formats that you can simply use instead of implementing them from scratch.

- For code, parsers check for syntax errors.

24.11 Short tour of data formats

- Python code:

  - Each statement is on a separate line

  - Changes in indentation are used to indicate entry/exit to blocks of code, e.g. within `def`, `for`, `if`, `while`...

- HTML: Basic structure is a mix of text with commands that are inside "tags" < ... >.

  Example:

```html
<html>
   <head>
      <title>HTML example for CSCI-100</title>
   </head>
   <body>
```

(continues on next page)

```
      This is a page about <a href="http://python.org">Python</a>.
      It contains links and other information.
   </body>
</html>
```

- Despite the clean formatting of this example, html is in fact free-form, so that, for example, the following produces exactly the same web page:

```
<html><head><title>HTML   example for CSCI-100</title>
</head> <body> This is a page about <a
href="http://python.org">Python</a>.  It contains   links
and other   information. </body> </html>
```

- JSON: used often with Python in many Web based APIs:

```
{
   "class_name": "CSCI 1100"
   , "lab_sections" : [
         { "name": "Section 01"
           , "scheduled": "T 10AM-12PM"
           , "location": "Sage 2704"
         }
         , { "name": "Section 02"
           , "scheduled": "T 12PM-2PM"
           , "location": "Sage 2112"
         } ]

 }
```

Similar to HTML, spaces do not matter.

- Json is a simple module for converting between a string in JSON format and a Python variable:

```
>>> import json
>>> x = ' [ "a", [ "b", 3 ] ] '
>>> json.loads(x)
['a', ['b', 3]]
```

24.12 Parsing ad-hoc data formats - Regular tabular data

We will examine some simple formats that you have already seen in various homeworks.

- Parsing files with fixed format in each line, delimited by a character

  Often used: comma (csv), tab or space

  - One example is a file of comma separated values, Each line has a label of a soup type and a number of cans we have available:

```
chicken noodle, 2
clam chowder, 3
```

  - Here is pseudo code for processing such files:

```
for each line of the file
  split into a list using the separator
  process the entries of the list into the desired form
```

- **Practice Problem:** write a simple parser for the soup list that returns a list of the form:

```
["chicken noodle", "chicken noodle", "clam chowder", "clam chowder", "clam chowder"]
```

24.13 Parsing ad-hoc data formats - Irregular tabular data

- Parsing files with one line per row of information, different columns containing unknown amount of information separated with a secondary delimiter

  - Example: Yelp data with the name of a restaurant, the lattitude, the longitude, the address, the URL, and a sequence of reviews

    Meka's Lounge|42.74|-73.69|407 River Street+Troy, NY 12180|http://www.yelp.com/biz/mekas-lounge-troy|Bars|5|2|4|4|3|4|5

  Information after column 5 are all reviews

  The address field is separated with a plus sign

  Pseudo code:

```
for each line of the file
    split using the separator
    split the entry with secondary separator
    for each value in the column
        handle value
```

- **Practice Problem:** Write a function that returns a list of lists for a file containing Yelp data. Each list should contain the name of the restaurant and the average review.

24.14 Summary

- You should now have enough information to easily write code to open, read, write and close files on local computers.

- Once text (or HTML) files found on the web are opened, the same reading methods apply just as though the files were local. Binary files such as images require special modules.

- Parsing a file requires understanding its format, which is, in a sense, the "language" in which it is written.

- You will practice with file reading and writing in future labs and homework assignments.

LECTURE 13 — EXERCISES

Solutions to the problems below must be sent to Submitty for grading. A separate file must submitted for each problem. Solutions must be submitted by 4 pm on Tuesday, October 17.

1. Given the file `census_data.txt`:

```
Line 1 |Location     2000      2011
Line 2 |New York State  18,976,811  19,378,102
Line 3 |New York City   8,008,686   8,175,133
Line 4 |
```

What is the output of the following code? (Note: the line numbers and the "|" are not actually in the file, they are just there to show that the contents of the 4 lines.)

```
f = open("census_data.txt")
line1 = f.readline()
line1 = line1.strip()
line2 = f.read()
line3 = f.readline()
print(line1)
print(line2)
print(line3)
f.close()
f = open("census_data.txt")
s = f.read()
line_list = s.split('\n')
print(len(line_list))
line_list = s.strip().split('\n')
print(len(line_list))
```

Submit your output as a text file.

2. Given a file containing test scores, one per line, we want to have a new file that contains the scores in increasing order. To do this, write a Python program that asks the user for two file name strings, one for the input scores and the second for the output, sorted scores. The program should open the first file (to read), read the scores, sort them, open the second file (to write), and output to this file the scores in increasing order. There should be one score per line, with the index on each line.

As an example, suppose the input file is `scores.txt` and it contains

```
75
98
75
100
21
```

```
66
83
15
```

then running your program should look like (in Wing IDE):

```
Enter the scores file: scores.txt
scores.txt
Enter the output file: scores_sorted.txt
scores_sorted.txt
```

When you look at the contents of `scores_sorted.txt` you should see

```
 0:  15
 1:  21
 2:  66
 3:  75
 4:  75
 5:  83
 6:  98
 7: 100
```

(Output the indices using two integer spaces {:2d} and the scores using three {:3d}.) You only need to submit the Python file. We will test with the example above and with a new file you have not seen.

LECTURE 14 — PROBLEM SOLVING AND DESIGN, PART 1

26.1 Overview

This is the first of our lectures dedicated primarily to problem solving and design rather than on particular programming constructs and techniques

- Design:
    - Choice of container/data structure; choice of algorithm.
        * At the moment, we don't know too many containers, but we will think about different ways to use the one container - lists - we do know about.
    - Implementation
    - Testing
    - Debugging
- We will discuss these in the context of several variations on one problem:
    - Finding the mode in a sequence of values — the value (or values) occuring most often.
- There is no direct connect to a chapter in the text.
- We will start with a completely blank slate so that the whole process unfolds from scratch. This includes looking for other code to adapt.
- Working through problems like this is a good way to review what we've learned thus far.

26.2 Problem: Finding the Mode

- Given a series of values, find the one that occurs most often.
- Variation 1: is there a limited, indexable range of values?
    - Examples that are consistent with this variation include test scores or letters of the alphabet
    - Examples not consistent include counting words and counting amino acids
- Variation 2: do we want just the modes or do we want to know how many times each value occurs?
- Variation 3: do we want a histogram where values are grouped?
    - Example: ocean temperature measurements, pixel intensities, income values.
    - In each of these cases, a specific value, the number of occurrences of a specific temperature, such as 2.314C, is not really of interest. More important is the number of temperature values in certain ranges.

26.3 Our Focus: A Sequence of Numbers

- Integers, such as test scores

- Floats, such as temperature measurements

26.4 Sequence of Discussion

- Brainstorm ideas for the basic approach. We'll come with at least two.
    - We will discuss an additional approach when we learn about dictionaries.
- Algorithm / implementation
- Testing
    - Generate test cases
    - Which test cases we generate will depend on the choice of algorithm. We will combine them.
- Debugging:
    - If we find a failed test case, we will need to find the error and fix it.
    - Use a combination of carefully reading the code, working with a debugger, and generating print statements.
- Evaluation
    - We can analyze using theoretical tools we will learn about later or through experimental timing

26.5 Discussion of Variations

- Frequency of occurrence:
    - What are the ten most frequently occurring values? What are the top ten percent most frequent values?
    - Output the occurrences for each value.
- Clusters / histograms:
    - Test scores in each range of 10
- Quantiles: bottom 25% of scores, median, top 25%

TWENTYSEVEN

LECTURE 15 — SETS

27.1 Overview

- Example: finding all individuals listed in the Internet Movie Database (IMDB)

- A solution based on lists

- Sets and set operations

- A solution based on sets.

- Efficiency and set representation

Reading is Section 11.1 of *Practical Programming*.

27.2 Finding All Persons in the IMDB file

- We are given a file extracted from the Internet Movie Database (IMDB) called `imdb_data.txt` containing, on each line, a person's name, a movie name, and a year. For example,

```
Kishiro, Yukito   | Battle Angel   | 2016
```

- Goal:

  - Find all persons named in the file

  - Count the number of different persons named.

  - Ask if a particular person is named in the file

- The challenge in doing this is that many names appear multiple times.

- First solution: store names in a list. We'll start from the following code, posted on Piazza in `lec15_find_names_start.py`, which is part of a Lecture 15 zip file.

```
imdb_file = input("Enter the name of the IMDB file ==> ").strip()
name_list = []
for line in open(imdb_file, encoding = "ISO-8859-1"):
    words = line.strip().split('|')
    name = words[0].strip()
```

and complete the code in class.

- The challenge is that we need to check that a name is not already in the list before adding it.

- You may access the data files and the starting code .py file from the Resources page of the Piazza site.

27.3 How To Test?

- The file `imdb_data.txt` has about 260K entries. How will we know our results are correct?

- Even if we restrict it to movies released in 2010-2012 (the file `imdb_2010-12.txt`), we still have 25K entries!

- We need to generate a smaller file with results we can test by hand

  - I have generated `hanks.txt` for you and will use it to test our program before testing on the larger files.

27.4 What Happens?

- Very slow on the large files because we need to scan through the list to see if a name is already there.

- We'll write a faster implementation based on Python *sets*.

- We'll start with the basics of sets.

27.5 Sets

- A Python set is an implementation of the mathematical notion of a set:

  - No order to the values (and therefore no indexing)

  - Contains no duplicates

  - Contains whatever type of values we wish; including values of different types.

- Python set methods are exactly what you would expect.

  - Each has a function call syntax and many have operator syntax in addition.

27.6 Set Methods

- Initialization comes from a list, a range, or from just `set()`:

```
>>> s1 = set()
>>> s1
set()
>>> s2 = set(range(0,11,2))
>>> s2
{0, 2, 4, 6, 8, 10}
>>> v = [4, 8, 4, 'hello', 32, 64, 'spam', 32, 256]
>>> s3 = set(v)
>>> s3
{32, 64, 4, 'spam', 8, 256, 'hello'}
```

- The actual methods are

  - `s.add(x)` — add an element if it is not already there

  - `s.clear()` — clear out the set, making it empty

  - `s1.difference(s2)` — create a new set with the values from `s1` that are not in `s2`.

    * Python also has an "operator syntax" for this:

```
s1 - s2
```

- – s1.intersection(s2) — create a new set that contains only the values that are in **both** sets. Operator syntax:

```
s1 & s2
```

- – s1.union(s2) — create a new set that contains values that are in either set. Operator syntax:

```
s1 | s2
```

- – s1.issubset(s2) —- are all elements of s1 also in s2? Operator syntax:

```
s1 <= s2
```

- – s1.issuperset(s2) — are all elements of s2 also in s1? Operator syntax:

```
s1 >= s2
```

- – s1.symmetric_difference(s2) — create a new set that contains values that are in s1 or s2 but **not in both**.

```
s1 ^ s2
```

- – x in s - evaluates to True if the value associated with x is in set s.
- We will explore the intuitions behind these set operations by considering
  - – s1 to be the set of actors in *comedies,*
  - – s2 to be the set of actors in *action movies*

  and then consider who is in the sets

```
s1 - s2

s1 & s2

s1 | s2

s1 ^ s2
```

27.7 Exercises

1. Sets should be relatively intuitive, so rather than demo them in class, we'll work through these as an exercise:

```
>>> s1 = set(range(0,10))
>>> s1


>>> s1.add(6)
>>> s1.add(10)


>>> s2 = set(range(4,20,2))
>>> s2
```

```
>>> s1 - s2


>>> s1 & s2


>>> s1 | s2


>>> s1 <= s2


>>> s3 = set(range(4,20,4))
>>> s3 <= s2
```

27.8 Back to Our Problem

- We'll modify our code to find the actors in the IMDB. The code is actually very simple and only requires a few set operations.

27.9 Side-by-Side Comparison of the Two Solutions

- Neither the set nor the list is ordered. We can fix this at the end by sorting.

    - The list can be sorted directly.

    - The set must be converted to a list first. The function `sorted` does this for us.

- What about speed? The set version is **MUCH FASTER** — to the point that the list version is essentially useless on a large data set.

    - We'll use some timings to demonstrate this quantitatively

    - We'll then explore why in the rest of this lecture.

27.10 Comparison of Running Times for Our Two Solutions

- List-based solution:

    - Each time before a name is added, the code — through the method `in` — scans through the entire list to decide if it is there.

    - Thus, the work done is proportional to the size of the list.

    - The overall running time is therefore roughly proportional to the `square` of the number of entries in the list (and the file).

    - Letting the mathematical variable N represent the length of the list, we write this more formally as $O(N^2)$, or "the order of N squared"

- Set-based code

- For sets, Python uses a technique called *hashing* to restrict the running time of the `add` method so that it is *independent of size of the set*.

  * The details of hashing are covered in CSCI 1200, Data Structures.

- The overall running time is therefore roughly proportional to the length of the set (and number of entries in the file).

- We write this as $O(N)$.

- We will discuss this type of analysis more later in the semester.

  - It is covered in much greater detail in Data Structures and again in Intro. to Algorithms.

27.11 Discussion

- Python largely hides the details of the containers — set and list in this case — and therefore it is hard to know which is more efficient and why.

- For programs applied to small problems involving small data sets, efficiency rarely matters.

- For longer programs and programs that work on larger data sets, efficiency does matter, sometimes tremendously. What do we do?

  - In some cases, we still use Python and choose the containers and operations that make the code most efficient.

  - In others, we must switch to programming languages, such as C++, that generate and use compiled code.

27.12 Summary

- Sets in Python realize the notion of a mathematical set, with all the associated operations.

- Operations can be used as method calls or, in many cases, operators.

- The combined core operations of finding if a value is in a set and adding it to the set are **much faster when using a set** than the corresponding operations using a list.

- We will continue to see examples of programming with sets when we work with dictionaries.

27.13 Extra Practice Problems

1. Write Python code that implements the following set functions using a combination of loops, the `in` operator, and the `add` function. In each case, `s1` and `s2` are sets and the function call should return a set.

   (a) `union(s1,s2)`

   (b) `intersection(s1,s2)`

   (c) `symmetric_difference(s1,s2)`

TWENTYEIGHT

LECTURE 15 — EXERCISES

Solutions to the problems below must be sent to Submitty for automatic scoring. A separate file must submitted for each problem. Solutions must be submitted by 4 pm on Tuesday, March 20.

1. What is the output of the following Python code? Write the answer by hand before you type it into the Python interpreter. One thing that will be hard to guess is the order of the sets, especially when the set mixes integers and strings. Therefore, you pretty much have to run the code to make sure you have the order right. (At the moment, our use of Submitty is not sophisticated enough to allow random order.) This means you get the points almost for free, but you should make the effort to be sure you understand what is happening.

```python
s1 = set([0,1,2])
s2 = set(range(1,9,2))
print('A:', s1.union(s2))

print('B:', s1)

s1.add('1')
s1.add(0)
s1.add('3')
s3 = s1 | s2
print('C:', s3)

print('D:', s3 - s1)
```

Note that this example does NOT cover all of the possible set operations. You should generate and test your own examples to ensure that you understand all of the basic set operations.

2. Write a Python program that asks the user for two strings: (1) the name of a file formatted in the same way as the IMDB data, and (2) a string that is the start of a last name. The program should output the number of different last names that are in the file and it should output the number of different names that start with the string. Your program *must* use a `set` and you may / are encouraged to start from the code written during lecture.

We define a last name to be everything up to the first comma in the name. (Some names will not have commas in them, and be careful to avoid adding empty last names to the set.) For example,

```
Downey Jr., Robert | Back to School |     1986
Downey Sr., Robert   | Moment to Moment  | 1975
Downey, Elsie    | Moment to Moment    | 1975
```

would result in three different last names, `Downey Jr.`, `Downey Sr.` and `Downey`.

Here is one example of running our solution:

```
Data file name: imdb/imdb_data.txt
Prefix: Down
48754 last names
10 start with Down
```

Before uploading your Python file to Submitty you should test using the data files (or restricted versions of them!) we provided for Lecture 15 on the Piazza page. On Submitty we will test with different files and examples.

LECTURE 16 — DICTIONARIES, PART 1

29.1 Overview

- More on IMDB

- Dictionaries and dictionary operations

- Solutions to the problem of counting how many movies are associated with each individual

- Other applications

29.2 How Many Movies is Each Person Involved In?

- Goals:

    - Count movies for each person.

    - Who is the busiest?

    - What movies do two people have in common?

- Best solved with the notion of a dictionary, but we'll at least consider how to use a list.

29.3 List-Based Solution — Straightforward Version

- Core data structure is a list of two-item lists, each giving a person's name and the count of movies.

- For example, after reading the first seven lines of our shortened `hanks.txt` file, we would have the list

```
[ ["Hanks, Jim", 3], ["Hanks, Colin", 1],
  ["Hanks, Bethan", 1], ["Hanks, Tom", 2] ]
```

- Just like our solution from the sets lectures, we can start from the following code:

```
imdb_file = input("Enter the name of the IMDB file ==> ").strip()
count_list = []
for line in open(imdb_file, encoding = "ISO-8859-1"):
    words = line.strip().split('|')
    name = words[0].strip()
```

- Like our list solution for finding all IMDB people, this solution is VERY slow — once again $O(N^2)$ ("order of N squared").

29.4 List-Based Solution — Faster Version Based on Sorting

- There is an alternate solution that would work for the number of unique names solution from lecture 15 as well. It is based on sorting.

- Append each name to the end of the list **without** checking if it is already there.

- After reading all of the movies, sort the entire resulting list

  - As a result, all instances of each name will now be next to each other.

- Go back through the list, counting the occurrence of each name

- This solution will be **much** faster than the first, but it is also more involved to write than the one we are about to write using dictionaries

29.5 Introduction to Dictionaries

- Association between "keys" (like words in an English dictionary) and "values" (like definitions in an English dictionary). The values can be **anything**.

- Examples:

```
>>> heights = dict()
>>> heights = {}      # either of these works
>>> heights['belgian horse'] = 162.6
>>> heights['indian elephant'] = 280.0
>>> heights['tiger'] = 91.0
>>> heights['lion'] = 97.0
>>> heights
{'belgian horse': 162.6, 'tiger': 91.0, 'lion': 97.0, 'indian elephant': 280.0}
>>> 'tiger' in heights
True
>>> 'giraffe' in heights
False
>>> 91.0 in heights
False
>>> list(heights.keys())
['belgian horse', 'tiger', 'lion', 'indian elephant']
>>> sorted(heights.keys())
['belgian horse', 'indian elephant', 'lion', 'tiger']
>>> heights.values()
dict_values([162.6, 91.0, 97.0, 280.0])
>>> list(heights.values())
[97.0, 162.6, 91.0, 280.0]
```

- Details:

  - Two initializations; either would work.

  - Syntax is very much like the subscripting syntax for lists, except dictionary subscripting/indexing uses keys instead of integers!

  - The keys, in this example, are animal species (or subspecies) names; the values are floats.

  - The in method tests only for the presence of the key, like looking up a word in the dictionary without checking its definition.

  - The keys are NOT ordered.

- Just as in sets, the implementation uses *hashing* of keys.
  - Conceptually, sets are dictionaries without values.

29.6 Lecture Exercise 1

You will have five minutes to work on the first lecture exercise.

29.7 Back to Our IMDB Problem

- Even though our coverage of dictionaries has been brief, we already have enough tools to solve our problem of counting movies.
- Once again we'll use the following as a starting point

```
imdb_file = input("Enter the name of the IMDB file ==> ").strip()
counts = dict()
for line in open(imdb_file, encoding = "ISO-8859-1"):
    words = line.strip().split('|')
    name = words[0].strip()
```

- The solution we give in class will output the counts for the first 100 individuals in alphabetical order. It will be up to you as an exercise to find the most frequently occuring individual.
- We will impose an ordering on the output by sorting the keys.
- We'll test first on our smaller data set and then again later on our larger ones.

29.8 Key Types

- Thus far, the *keys* in our dictionary have been strings.
- Keys can be any "hashable" type — string, int, float, booleans, tuples.
  - Lists, sets and other dictionaries can not be keys.
- Strings are by far the most common key type
- We will see an example of integers as the key type by the end of the Lecture 17 (next set of) notes.
- Float and boolean are general poor choices. Can you think why?

29.9 Value Types

- So far, the *values* in our dictionaries have been integers and floats.
- But, any type can be the values
  - boolean
  - int
  - float
  - string

- – list

- – tuple

- – set

- – other dictionaries

- Here is an example using our IMDB code and a set:

```
>>> people = dict()
>>> people['Hanks, Tom'] = set()
>>> people['Hanks, Tom'].add('Big')
>>> people['Hanks, Tom'].add('Splash')
>>> people['Hanks, Tom'].add('Forest Gump')
>>> print(people['Hanks, Tom'])
{'Forest Gump', 'Big', 'Splash'}
```

- Here is another example where we store the continent and the population for a country instead of just the population:

```
countries = dict()
countries.clear()
countries['Algeria'] =  (37100000, 'Africa')
countries['Canada'] = (34945200, 'North America' )
countries['Uganda'] = (32939800, 'Africa')
countries['Morocco'] = (32696600, 'Africa')
countries['Sudan'] = (30894000, 'Africa')
```

- We access the values in the entries using *two consecutive subscripts*. For example,

```
name = "Canada"
print("The population of {} is {}".format(name, countries[name][0]))
print("It is in the continent of", countries[name][1])
```

29.10 Removing Values: Sets and Dictionaries

- For a set:
    - `discard` removes the specified element, and does nothing if it is not there
    - `remove` removes the specified element, but fails (throwing an exception) if it is not there
- For a dictionary, it is the `del` function.
- For both sets and dictionaries, the `clear` method empties the container.
- We will look at toy examples in class

29.11 Other Dictionary Methods

- The following dictionary methods are useful, but not so much as the ones we've discussed.
    - `get`
    - `pop`
    - `popitem`

    – update

- Use the `help` function in Python to figure out how to use them and to find other dictionary methods.

29.12 Summary of Dictionaries

- Associate "keys" with "values"

- Feels like indexing, except we are using keys instead of integer indices.

- Makes counting and a number of other operations simple and fast.

- Keys can be any "hashable" value, usually strings, sometimes integers.

- Values can any type whatsoever.

29.13 Additional Practice

1. Write a function that takes the IMDB dictionary — which associates strings representing names with integers representing the count of movies — and an integer representing a `min_count`, and removes all individuals from the dictionary involved in fewer than `min_count` movies.

LECTURE 16 — EXERCISES

Solutions to the problems below must be sent to Submitty for automatic scoring. A separate file must submitted for each problem. Solutions must be submitted by 4 pm on Tuesday, March 21.

1. Write a Python program that forms a dictionary called `countries` that associates the population with each of the following countries (in millions):

   - Algeria 37.1

   - Canada 3.49

   - Uganda 32.9

   - Morocco 32.7

   - Sudan 30.9

   - Canada 34.9 # a correction to the error above.

   and then prints the length of the `countries` dictionary, the sorted list of the keys in `countries` and a sorted list of the values in `countries`. There should be six assignment statements in your program (seven if you include initializing the dictionary) and three lines of output from your program. Please initialize your dictionary using `dict()` rather than `{}`

2. Our solution to the IMDB problem thus far has not actually told us who is the busiest individual in the Internet movie database. Your job in this part is to complete this task. Starting from the code produced in class, which will be immediately posted on the course website (in the *Code written in class* area), write a program that finds and prints the name of the individual who appears the most times in the IMDB file you are given. Also, count and output the number of individuals who appear only 1 time in the IMDB.

   For example if the answer were `Thumb, Toni` and this person had appeared 100 times, and if 2,000 people had only appeared once, then your output should be

   ```
   Enter the name of the IMDB file ==> imdb_data.txt
   Thumb, Toni appears most often: 100 times
   2000 people appear once
   ```

   We strongly suggest that you test your solution on the `hanks.txt` dataset first! We will test on multiple files. You do not need to worry about the possibility of a tie for the most commonly occuring name. Please initialize your dictionary using `dict()` rather than `{}`

LECTURE 17 — DICTIONARIES, PART 2

31.1 Overview

- Recap
- More IMDB examples:
    - Dictionaries of string/set pairs
    - Converting dictionaries with one key to another
    - Combining information from multiple dictionaries
- A different view of dictionaries: storing attribute/value pairs.
- Accessing APIs and getting data back as a dictionary.

31.2 Recap of dictionaries

- On the surface, dictionaries look like lists, except, you can have anything for indices (keys), not just numbers starting with 0.
- The following two store the same information:

```
>>> listoption = ['a','b','c']
>>> dictoption = {0:'a', 1: 'b', 2:'c'}
```

Note that this is a new way for us to initialize a dictionary.

- You would access them in the same way:

```
>>> listoption[1]
'b'
>>> dictoption[1]
'b'
```

- You would update them in the same way:

```
>>> listoption[1] = 'd'
>>> dictoption[1] = 'd'
```

- But you can't extend them in the same way. For example:

```
>>> listoption[10] = 'e'
```

is illegal, but:

```
>>> dictoption[10] = 'e'
```

is perfectly fine. Be sure you can explain why.

- Of course the power of a dictionary is that keys can be anything, or at least anything *hashable*

```
>>> d = {'Gru':3, 'Margo':4}
>>> d['Gru']
3
```

- This dictionary has strings as keys and integers as values. The values can really be anything:

```
>>> d2 = {'Gru': set( [123,456] ), 'Margo': set( [456] ) }
>>> d2['Gru']
{456, 123}
```

- Note that since keys can be anything, we need to know how to print or iterate through the values in a dictionary. This is actually quite trivial using a dictionary:

```
>>> d2.keys()
dict_keys(['Gru', 'Margo'])
>>> list(d2.keys())
['Gru', 'Margo']
>>> for key in d2:     # or, for key in d2.keys():
...     print(key, d2[key])
Gru {[123, 456]}
Margo {[456]}
```

31.3 Copying and Aliasing Dictionaries

- We'll take a few minutes in class for you to try to predict the output of the following:

```
d = dict()
d[15] = 'hi'
L = []
L.append(d)
d[20] = 'bye'
L.append(d.copy())
d[15] = 'hello'
del d[20]
print(L)
```

- The result may surprise you, but it reflects the difference between making an alias to an object and making a full copy of an object.

  - An alias is also sometimes known as a *shallow copy*

  - A full copy is also sometimes known as a *deep copy*

- Assignment between lists, between sets, and between dictionaries all involve aliasing / shallow copying!

- At this point we will take a few minutes for you to work on the first lecture exercise.

31.4 Back to IMDB: Dictionaries Whose Values Are Sets

- In our IMDB data, an individual may be listed more than once for each movie. For example, Tom Hanks is listed six times for *Polar Express*

- In order to determine who was involved in the most different movies, we need to keep a set of movies for each individual instead of a count.

- We will modify our solution to the IMDB example to find out who was involved in the most different movies.

    - The solution will be posted on the course website.

31.5 Converting to Other Dictionaries: The N Busiest Individuals

- Suppose we want to find the top 10 or 25 busiest individuals in the IMDB, based on the number of different movies they are involved in.

- Now we need a different dictionary:

    - Keys are integers representing the number of movies

    - Values are lists of actors.

        * Why don't we need sets here?

- We will show how to extend our code to build this dictionary from our original dictionary.

- Next, we will need to access the keys from this dictionary in reverse order and print out names of the individuals, stopping when we've printed the top N busiest (allowing more in the case of ties).

31.6 More That We Can Do With the IMDB

- We now have an actors dictionary whose keys are actor names and whose values are sets of movies.

- We can also construct a different dictionary whose keys are movies and whose values are sets of actors.

- Using this we can find all sorts of information:

    - What movie involved the most people?

    - How many different people have been in movies that included Meryl Streep?

    - Solve the "degrees of Kevin Bacon" problem:

        1. Who has been in a movie with Kevin Bacon? These people are degree 1.

        2. Who is not a degree 1 individual and has been in a movie with a person who was in a movie with Kevin Bacon? These people are degree 2 individuals.

        3. Who is not a degree 1 or 2 individual, and has been in a movie with a degree 2 individual (in a movie with a person who has been in a movie with a person who was in a movie with Kevin Bacon)? These people are degree 3 individuals.

        4. Etc.

31.7 Attribute / Value Pairs

- We can use dictionaries to construct even more complicated data structures: dictionaries as values, lists of dictionaries, etc.

- Consider the problem of representing all the houses a real estate company is trying to sell.

- We could keep a list with information about each property, but a list of what?

- We will look at describing each house as a dictionary, with the keys being the "attributes", and the values being, well, the values of the attributes.

- Examples include the listing reference number, the address, the number of bedrooms, the price, whether or not it has a pool, the style of the house, the age, etc.
  - Some properties will not be known and therefore they will not be represented in the dictionary.

- We will work through a made-up example in class, producing a list of dictionaries. This list will be called `houses`.

- As an exercise you can think about write coding that finds all houses in our house list that have at least 4 bedrooms (attribute is `bedrooms`, value is an integer), a pool (attribute is `pool`, value a string describing if the pool is above ground or below), for a price below $300,000 (atttribute is `price`, value is an int).

- Overall, this a simple Python implementation of the storage and access of information in a *database*.

31.8 Accessing APIs

- Many APIs (Application Programming Interfaces) accessible on the internet return values that are **JSON** (Java Script Object Notation) strings. These are easily loaded into Python objects, often involving dictionaries.

- The best way to understand the dictionary structure returned by an API is to seek documentation. If that fails, you can print the top level keys and values to explore.

- Public APIs, which do not not require authentication, are accessed as follows:

```python
import urllib.request
import json

url = "enter your public url here"
f = urllib.request.urlopen(url)
rawcontent = f.read()
content = json.loads(rawcontent.decode("utf-8"))
```

- An example of a public API (used in our image lab):
  - **nominatim** gives you a bounding box of geolocation for a given location. Let's see this for 'Troy, NY':

```python
url = "http://nominatim.openstreetmap.org/"\
      "search?q={}&format=json&polygon_geojson=1&addressdetails=0"\
      .format('Troy,NY')
```

- Many sources require authentication with an API key through the `oauth2` authentication module. But, the overall method of access remains the same after authentication.

- Once we understand the structure, we can write code to extract the information we want.

31.9 Final Example

1. Given the following dictionary for hobbies for people:

```
hobby = {'Gru':set(['Hiking','Cooking']), 'Edith':set(['Hiking','Board Games'])}
```

   create a new dictionary that lists people for each hobby:

```
{'Hiking': {'Gru','Edith'}, 'Cooking':{'Gru'}, 'Board Games':{'Edith'}}
```

31.10 Summary

- Dictionaries of sets.

- Dictionaries where the keys are numbers.

- A variety of examples to extract information from the IMDB data set.

- Dictionaries as database — storing attribute / value pairs.

- Accessing information from public APIs

31.11 Additional Dictionary Practice Problems

1. Create a dictionary to store the favorite colors of the following individuals

   - Thomas prefers red

   - Ashok prefs green

   - Sandy prefers red

   - Alison prefers orange

   - Fei prefers green

   - Natasha prefs blue

   Then add some others of your own. Now, write code to change Fei's preference to green and to remove Sandy's preference from the dictionary.

2. Using the dictionary from the first problem, write code to find which color is most commonly preferred. Use a second dictionary, one that associates strings (representing the colors) with the counts. Output the most common color. If there are ties, output all tied colors.

3. Complete the fast, list solution to the movie counting problem based on sorting, as outlined at the start of the lecture notes.

4. Write a program that uses a dictionary that associates integers (the key) and sets strings (the values) to find the number of movies in each year of the IMDB. Start from any of the IMDB examples. Write additional code that uses the `years_and_movies` dictionary to find the year that has the most movies.

5. Use a dictionary to determine which last names are most common in the IMDB data we have provided. Count individual people not the movies they appear in. For example, `'Hanks, Tom'` counts as one instance of the name `'Hanks'` despite the fact that he is in many movies. Assume that the last name ends with the first `','` in the actual name. Start this problem by thinking about what the dictionary keys and values should be.

6. Which two individuals have the most movies in common? To solve this you will need to start from the dictionary that associates each individual with the set of movies s/he is involved in. Then you will need double for loops. At first glance this appears that it might be very, very slow, but it can be made much faster by intelligently terminating loops. To illustrate, if you find a pair of individuals with k movies in common then you never have to even consider an individual involved in fewer than k movies!

LECTURE 17 — EXERCISES

Solutions to the problems below must be sent to Submitty for automatic scoring. A separate file must submitted for each problem. Solutions must be submitted by 4 pm on Friday, March 30.

1. What is the output of the following code?

```python
d1 = dict()
l1 = [ 5, 6, 7 ]
d1['car'] = l1
d1['bus'] = l1.copy()
l1.append( [8,9] )
d1['truck'] = d1['bus']
d1['bus'].append(10)
d1['truck'].pop(0)
print("list:", l1)
for v in sorted(d1.keys()):
    print("{}: {}".format( v, d1[v] ))
```

2. Write a Python program that finds the name of the movie in our Internet Movie Database that involved the most *unique* individuals. Print the name of the movie on one line and the number of individuals on the next line. You do not need to consider the possibility of ties and you should assume all actors and movies have the correct capitalization. Finally, print the number of movies involving only one individual (yes, there are such movies). For example, if the name of the movie is *Ben Hur* with 2,342 individuals, and 165 movies have only one individual then the output from your program will look like:

```
Enter the name of the IMDB file ==> imdb_data.txt
2342
Ben Hur
165
```

Note that you only need to create one dictionary for this exercise. Start by planning the organization of this dictionary and thinking through how you are going to use it. As with Lecture Exercises 16, make sure you use *dict()* to initialize your empty dictionary.

LECTURE 18 — CLASSES, PART 1

33.1 Overview

- Define our own types and associated functions

- Encapsulate data and functionality

- Raise the "level of abstraction" in our code

- Make code easier to write and test

- Reuse code

33.2 Potential Examples

In each of these, think about what data you might need to store to represent the "object" and what functionality you might need to apply to the data.

- Date

- Time

- Point

- Rectangle

- Student

- Animal

- Molecule

33.3 An Example from Earlier in the Semester

- Think about how difficult it was to keep track of the information about each restaurant in the Yelp data.

- You had to:

    - Remember the indices of (a) the restaurant name, (b) the latitude and longitude, (c) the type of restaurant, (d) the address, etc.

    - Form a separate list inside the list for the ratings.

    - Write additional functions to exploit this information

- If we used a class to represent each restaurant:

– All of the information about the restaurant would be stored and accessed as named attributes

– Information about the restaurants would be accessed through functions that we write for the class.

33.4 Point2d Class

• Simplest step is to just tell Python that `Point2d` will exist as a class, deferring the addition of information until later.

```python
class Point2d(object):
    pass
```

• The Python reserved word `pass` says that this is the end of the class definition.

– We will not need this later when we put information into the class.

33.5 Attributes

• Classes do not get interesting until we put something in them.

• The first thing we want is variables so that we can put data into a class.

– In Python these variables are often called *attributes*.

– Other languages call them *member variables*.

• We will see three different ways to specify attributes.

33.6 Assigning Attributes to Each Instance

• Points have an `x` and a `y` location, so we can write, for example,

```python
from math import sqrt
p = Point2d()
p.x = 10
p.y = 5
dist_from_origin = sqrt(p.x**2 + p.y**2)
```

• We have to do this for each class instance.

• This is prone to mistakes:

– Could forget to assign the attributes

– Could accidentally use different names for what is intended to be the same attribute.

• Example of an error

```python
q = Point2d()
q.x = -5
dist_from_origin = sqrt(q.x**2 + q.y**2)    # q.y does not exist
```

33.7 Defining the Attributes Inside the Class

- The simplest way to make sure that all variables that are instances of a class have the appropriate attributes is to define them inside the class.

- For example, we could redefine our class as

```
class Point2d(object):
    x = 0
    y = 0
```

- All instances of `Point2d` now have two attributes, x and y, and they are each initialized to 0.

- We no longer need the `pass` because there is now something in the class.

33.8 Defining the Attributes Through An Initializer / Constructor

- We still need to initialize x and y to values other than 0:

```
p = Point2d()
p.x = 10
p.y = 5
```

- What we'd really like to do is initialize them at the time we actually create the `Point2d` object:

```
p = Point2d(10,5)
```

- We do this through a special function called an *initializer* in Python and a *constructor* in some other programming languages.

- Inside the class this looks like

```
class Point2d(object):
    def __init__( self, x0, y0 ):
        self.x = x0
        self.y = y0
```

- Our code to create the point now becomes

```
p = Point2d(10,5)
```

- Notes:

  – Python uses names that start and end with two `'_'` to indicate functions with special meanings. More on this later in the lecture.

  – The name `self` is special notation to indicate that the object itself is passed to the function.

- If we'd like to initialize the point to (0,0) without passing these values to the constructor every time then we can specify default arguments

```
class Point2d(object):
    def __init__( self, x0=0, y0=0 ):
        self.x = x0
        self.y = y0
```

allowing the initalization

```
p = Point2d()
```

33.9 Methods — Functions Associated with the Class

- We create functions that operate on the class objects inside the class definition:

```python
import math

class Point2d(object):
    def __init__( self, x0, y0 ):
        self.x = x0
        self.y = y0

    def magnitude(self):
        return math.sqrt(self.x**2 + self.y**2)

    def dist(self, o):
        return math.sqrt( (self.x-o.x)**2 + (self.y-o.y)**2 )
```

  these are called *methods*

- This is used as

```python
p = Point2d(0,4)
q = Point2d(5,10)
leng = q.magnitude()
print("Magnitude {:.2f}".format( leng ))
print("Distance is {:.2f}".format( p.dist(q)))
```

- The method `magnitude` takes a single argument, which is the `Point2d` object called `self`. Let's examine this:

  - The call `q.magnitude()` appears to have no arguments, but when Python sees this, it turns it into its equivalent:

```python
Point2d.magnitude(q)
```

    which is completely legal Python syntax.

  - The name `self` is not technically special in Python, but it is used by convention to refer to the object that the method is "called upon". This is q' in the call `q.magnitude()`

- The method `dist` takes two `Point2d` objects as arguments. The example call

```python
p.dist(q)
```

  becomes

```python
Point2d.dist(p,q)
```

  so now argument p maps to parameter `self` and argument q maps to parameters o

33.10 Lecture Exercises, Part 1

Our lecture exercises for today will involve adding to the `Point2d` class and testing it. Make sure you have downloaded the `Point2d.py` file from the Piazza site.

We will allow some time to work on the first lecture exercise.

33.11 Operators and Other Special Functions

- We'd like to write code that uses our new objects in the most intuitive way possible.

- For our point class, this involves use of operators such as

```
p = Point2d(1,2)
q = Point2d(3,5)
r = p+q
s = p-q
t = -s
```

- Notice how in each case, we work with the `Point2d` variables (objects) just like we do with int and float variable (objects).

- We implement these by writing the special functions `__add__`, `__sub__`, and `__neg__`

- For example, inside the `Point2d` class we might have

```
def __add__(self,other):
    return Point2d(self.x + other.x, self.y+other.y)
```

Very important: this creates a new `Point2d` object.

- When Python sees p+q, it turns it into the function call

```
Point2d.__add__(p,q)
```

which is exactly the syntax of the function definition we created.

- We have already seen this with operators on integers and strings. As examples,

```
5+6
```

is equivalent to

```
int.__add__(5,6)
```

and

```
str(13)
```

is equivalent to

```
int.__str__(13)
```

- Implicit in this discussion is the notion that `int` is in fact a class in Python. The same is true of `str` and `float` and `list`.

- Note that we can also define boolean operators such as `==` and `!=` through the special functions `__eq__` and `__neq__`

33.12 Classes and Modules

- Each class should generally be put into its own module, or several closely-related classes should be combined in a single module.

  – We are already doing this with `Point2d`.

- Doing so is good practice for languages like C++ and Java, where classes are placed in separate files.

- Testing code can be included in the module or placed in a separate module.

- We will demonstrate this in class and post the result on the course website.

33.13 More Lecture Exercises

At this point we will stop and take a bit of time to work on the next part of the lecture exercises.

33.14 When to Modify, When to Create New Object

- Some methods, such as `scale`, modify a single `Point2d` object

- Other methods, such as our operators, create new `Point2d` objects without modifying existing ones.

- The choice between this is made on a method-by-method basis by thinking about the meaning — the *semantics* — of the behavior of the method.

33.15 Programming Conventions

- Don't create attributes outside the class.

- Don't directly access or change attributes except through class methods.

  – Languages like C++ and Java have constructions that enforce this.

  – In languages like Python it is not a hard-and-fast rule.

- Class design is often most effective by thinking about the required methods rather than the required attributes.

  – As an example, we rarely think about how the Python `list` and `dict` classes are implemented.

33.16 Time Example

- In the remainder of the lecture, we will work through an extended example of a `Time` class

- By this, we mean the time of day, measured in hours, minutes and seconds.

- We'll brainstorm some of the methods we might need to have.

- We'll then consider several different ways to represent the time internally:

  – Hours, minutes and seconds

  – Seconds only

- – Military time

- Despite potential internal differences, the methods — or at least the way we call them — will remain the same

  - – This is an example of the notion of *encapsulation*, which we will discuss more in Lecture 19.

- At the end of lecture, the resulting code will be posted and tests will be generated to complete the class definition.

33.17 Summary

- Define new types in Python by creating classes

- Classes consist of *attributes* and *methods*

- Attributes should be defined and initialized through the special method call `__init__`. This is a *constructor*

- Other special methods allow us to create operators for our classes.

- We looked at a *Point2d* and *Time* example.

LECTURE 18 — EXERCISES

Solutions to the problems below must be sent to Submitty for automatic scoring. A separate file must submitted for each problem. Solutions must be submitted by 4 pm on Tuesday, April 4.

1. Starting from the `Point2d.py` file you download from the Resources page of the Piazza site, please do the following:

   (a) Write a new `Point2d` method called `scale` that takes as an input argument a `Point2d` object (`self`) and a numerical value (int or float) and multiples both the `x` and `y` attributes by this value.

   (b) Write a new `Point2d` method called `dominates` that two takes two `Point2d` objects and returns `True` if and only if the x coordinate of the first object is greater than that of the second object and the y coordinate of the first object is greater than that of the second object.

   (c) The code to test these functions is commented out in the main code area. Please remove this commenting, test on Piazza, and submit your resulting `Point2d.py` file. Call it `Point2d_q1.py`

2. Copy your resulting file from the first question to a new file, perhaps called `Point2d_q2.py`.

   (a) Write and test the implementation of the method `__str__` which returns a string created from the values of a `Point2d` object. For our purposes this is mostly used to create a string that can be printed. Make sure you have this working before you proceed to the other parts of this exercise because they depend on it.

   (b) Write the implementation of the subtraction method `__sub__` for the `Point2d` object. Uncomment the code in the main area and test this in Wing IDE.

   (c) Write the implementation of the method `__mul__` which is like the `scale` function you wrote for part 1, but it creates a new `Point2d` object.

   (d) Write the implementation of the method `__eq__` which returns `True` if and only if the two `Point2d` objects have exactly the same x and y values.

   For each of these you should look at the commented out main code in the `Point2d.py` file you were provided to see how these methods should be used. Uncomment this code, test your methods, and upload to Submitty when you are done.

LECTURE 19 — CLASSES, PART 2

35.1 Overview

- Review of classes
- Revisiting our Yelp data: a `Restaurant` class.
- Techniques that we will see:
  - Calling class methods from within the class
  - Class objects storing other objects, such as lists
  - Lists of class objects

35.2 Review of Classes

We will use our `Point2d` class solution from Lecture 18 to review the following:

- Attributes:
  - These store the data associated with each class instance.
  - They are usually defined inside the class to create a common set of attributes across all class instances.
- Initialization: function `__init__` called when the object is created.
  - Should assign initial values to all attributes
- Methods
  - Each includes the object, often referred to as `self`, as the first argument.
  - Some change the object, some create new objects
- Special methods start and end with two underscores. Python interprets their use in a variety of distinct ways:
  - `__str__` is the string conversion function
  - `__add__`, `__sub__`, etc. become operators
- Each of these special methods builds on the "more primitive" methods

35.3 Larger Example — Restaurant Class

Recall Lab 5 on the Yelp data:

- Read and parse input lines that look like:

```
The Greek House|42.73|-73.69|27 3rd St+Troy, NY 12180|\
    http://www.yelp.com/biz/the-greek-house-troy|Greek|1|5|4|5|4|4|5|5|5|5|5|4
```

- Find restaurants and print out information based on a user selection

- Original implementation based on a list was awkward:
    - We had to remember the role of each index of the list — 0 was the name, 1 was the latitude, etc.

- New implementation here is based on a class

35.4 Start to a Solution, the Main Code

Let's look at `lec19_restaurants_exercise.py`, downloadable as part of the Lecture_19 zip file:

- This is the code that *uses* the `Restaurant` class.
    - We start by considering how the class will be used rather than how we write it.

- Main function to initialize a restaurant is called `convert_input_to_restaurant`
    - Parses a restaurant line
    - Creates and returns a `Restaurant` object

- Function `build_restaurant_list`
    - Opens the input file
    - Reads each line
    - Calls `convert_input_to_restaurant`, and appends the resulting restaurant to the back of a list

- Main code:
    - Builds the restaurant list
    - Prints the first three restaurants in the list
    - Includes commented-out code that
        * Gets the name of a city
        * Finds the restaurant with the highest average rating

    We will complete this code soon.

35.5 Functionality Needed in the Restaurant Class

- Some functionality is determined by reading the code we have already written
    - Includes both methods and attributes

- Add other functionality by considering the methods that must be in the `Restaurant` class, including the parameters that must be passed to each method.

- Add attributes last...

35.6 Turning to the Actual Restaurant Class

Look at `Restaurant.py` which was distributed with the Lecture_19 files.

- The `__init__` function specifies the attributes.
  - Other attributes could be added, such as the average rating, but instead these are computed as needed by methods.
  - Importantly, each class object stores a list of ratings, illustrating the fact that classes can store data structures such as lists, sets, and dictionaries.
- The `Restaurant` class has more complicated attributes than our previous objects
  - `Point2d` object,
  - A list for the address entries
  - A list of scores
- There is nothing special about working with these attributes other than they "feel" more complicated.
  - Just apply what you know in using them
  - Our lecture exercises will help

35.7 In-Class Example

Together we will add the following two methods `Restaurant` to get our demonstration example to work:

1. The `is_in_city` method
2. The `average_review` method

35.8 Discussion

- What is not in the `Restaurant` class?
  - No input or line parsing. Usually, we don't want the class tied to the particular form of the input.
  - As an alternative, we could add a method for each of several different forms of input.
- Often it is hard to make the decision about what should be inside and what should be outside the class.
  - One example the method we wrote to test if restaurant is in a particular city. As an alternative we could have written a different method that returns that name of the city and make the comparison outside the class.
- We could add an `Address` class:
  - Reuse for objects other than restaurants
  - Not needed in this (relatively) short example.
  - More flexible than our use of a list of strings from an address line.

35.9 Summary

- Review of the main components of a Python class:
  - Attributes
  - Methods
  - Special methods with names starting and ending with __
    * Initializer method is most important
- Important uses of Python classes that we have seen today:
  - Classes containing other objects as attributes
  - Lists of class objects.
- Design of Python classes
  - Start by outlining how they are to be used
  - Leads to design of methods
  - Specification of attributes and implementation of methods comes last

LECTURE 19 — EXERCISES

Solutions to the problems below must be sent to Submitty for automatic scoring. A separate file must submitted for each problem. Solutions must be submitted by 4 pm on Friday, April 7.

1. The `Restaurant.py` file you were provided includes a main code area with code to test functions that have not yet been written (see the methods with `pass` as their only statements). These are the `min_review`, the `max_review`, the `latitude` and the `longitude`. Please implement these functions, test them, and submit the resulting `Restaurant.py`.

2. Copy the code in `lec19_restaurant_exercise.py` into a file called `lec19_american_in_troy.py`. Rewrite the main code in this new file to list the names of all Restaurants in *Troy* that have *American* in their category and that have an average rating of **more than** 3.0. The **only output** should be the alphabetical list of restaurant names, one per line of output. The trick is that you are *not allowed to change* the `Restaurant` class at all. This will require that you access and use both one or two methods from `Restaurant` and some of its attributes directly. Upload your `lec19_american_in_troy.py` to Submitty when you are done. Submitty will use our `Restaurant.py` file to test.

LECTURE 20 — SEARCHING

37.1 Overview

- Notion of an algorithm
- Problems:
    - Finding the two smallest values in a list
    - Finding the index of a particular value in a list
    - Finding the index of a particular value in a **sorted** list
    - Sorting a list (Lecture 21)
- Analyzing our solutions:
    - Mathematically
    - Experimental timing

Material for Lectures 20 and 21 is in Chapters 12 and 13 of the text.

37.2 Algorithm

- Precise description of the steps necessary to solve a computing problem
- Description is intended for people to read and understand
- Gradual refinement:
    - Starts with English sentences
    - Gradually, the sentences are made more detailed and more like programming statements
    - Allows us to lay out the basic steps of the program before getting to the details.
- A program is an *implementation* of one or more algorithms.

37.3 Multiple Algorithms

- Often there are many different algorithms that can solve a problem.
- They differ in:
    - Ease of understanding

– Ease of implementation

– Efficiency

- All three considerations are important and their relative weight depends on the context.

37.4 Problem 1: Finding the Two Smallest Values in a List

- Given a list of integers, floats, or any other values that can be compared with a less than operation, find the two smallest values in the list AND their indices in the list

- We need to be careful with this problem formulation: are duplicates allowed? does it matter?

37.5 Brainstorming Session

1. Outline two or more approaches to finding the indices of the two smallest values in a list.

2. Think through the advantages and disadvantages of each approach.

3. Write a more detailed description of the solutions.

4. How might your approaches change if we just have to find the values and not the indices?

37.6 Evaluating Our Solutions Analytically

We've already covered this briefly in Lecture 15.

- Count the number of steps as a function of the size of the list.

  – Usually we use N as a variable to indicate this size.

- Informally, if the number of operations is (roughly) proportional to N we write $O(N)$ (read as "order of N")

- If the number of operations is proportional to $NlogN$ we write $O(NlogN)$.

  – Importantly, the best sorting algorithms, including the one implemented in Python for lists, are $O(NlogN)$.

- We will informally apply this analysis to our solution approaches.

37.7 Evaluating Our Solutions Experimentally

- Needs:

  – generate example data, and

  – time our algorithm implementations.

- Experimental data can be generated using the `random` module. We will make use of

  – `randrange`

  – `shuffle`

- Timing uses the `time` module and its `time` function, which returns the number of seconds (as a float) since an arbitrary start time called an "epoch".

– We will compute the difference between a start time and an end time as our timing measurement.

37.8 Completing the Solutions

- We will implement two of the algorithms we came up with to find the indices of the two smallest values in the list:

```python
import random
import time

def index_two_v1( values ):
    pass  # not implemented yet




def index_two_v2( values ):
    pass # not implemented yet




if __name__ == "__main__":
    n = int(input("Enter the number of values to test ==> "))
    values = list(range(0,n))
    random.shuffle( values )

    s1 = time.time()
    (i0,i1) = index_two_v1(values)
    t1 = time.time() - s1
    print("Ver 1:  indices ({},{}); time {:.3f} seconds".format(i0,i1,t1))

    s2 = time.time()
    (j0,j1) = index_two_v2(values)
    t2 = time.time() - s2
    print("Ver 2:  indices ({},{}); time {:.3f} seconds".format(j0,j1,t2))
```

We will experiment with these implementations.

37.9 Searching for a Value

- Problem: given a list of values, L, and given a single value, x, find the (first) index of x in L or determine that x is not in L.

- Basic algorithm is straightforward, and requires $O(N)$ steps

- We can solve this in Python using a combination of in and index, or by writing our own loop.

  – The text book discusses a number of variations on the algorithm.

- We will implement our own variation as an exercise.

37.10 Binary Search

- If the list is **ordered**, do we have to search it by looking at location 0, then 1, then 2, then 3, ...?

- What if we looked at the middle location first?

    - If the value of x is greater than that value, we know that the first location for x is in the **upper half of the list**.

    - Otherwise, the first location for x is in the **lower half** of the list

- In other words, by making one comparison, we have eliminated half the list in our search!

- We can repeat this process of "halving" the list until we reach just one location.

37.11 Algorithm and Implementation

- We need to keep track of two indices:

    - `low`: all values in the list at locations 0..`low-1` are less than x

    - `high`: all values in the list at locations `high` ..N are greater than or equal to x. Write N as the length of the list.

- Initialize `low = 0` and `high = N`.

- In each iteration of a while loop

    - Set `mid` to be the average of `low` and `high`.

    - Update the value of `low` or `high` based on comparing x to `L[mid]`.

- Here is the actual code:

```
def binary_search( x, L):
    low = 0
    high = len(L)
    while low != high:
        mid = (low+high)//2
        if x > L[mid]:
            low = mid+1
        else:
            high = mid
    return low
```

37.12 Practice

1. Using

```
L = [ 1.3, 7.9, 11.2, 15.3, 18.5, 18.9, 19.7 ]
```

what are the values of `low`, `high` and `mid` each time through the while loop for the calls

```
binary_search( 11.2, L )
```

```
binary_search( 19.1, L )



binary_search( -1, L)




binary_search( 25, L)
```

2. How many times will the loop execute for $N = 1,000$ or $N = 1,000,000$? (You will not be able to come up with an exact number, but you should be able to come close.) How does this compare to the linear search?

3. Would the code still work if we changed the > to the >=? Why?

4. Modify the code to return a tuple that includes both the index where x is or should be inserted and a boolean that indicates whether or not x is in the list.

We will also perform experimental timing runs if we have time at the end of class.

37.13 Summary

- Algorithm vs. implementation
- Criteria for choosing an algorithm: speed, clarity, ease of implementation
- Timing/speed evaluations can be either analytical or experimental.
- Searching for indices of two smallest values
- Linear search
- Binary search of a list that is ordered.

LECTURE 20 — EXERCISES

Solutions to the problems below must be sent to Submitty for automatic scoring. A separate file must submitted for each problem. Solutions must be submitted by 4 pm on Tuesday, April 11.

For both of these exercises download the file `lec20_ex.zip` from the course Piazza Resource page. It includes a data file and `lec20_ex_start.py`. You start from the latter for both exercises. You will notice that the main code in this file requests the name of a data file from the user, opens and reads the file to form a list of values, and then in a loop requests several different values to search for, outputting the result of the function call for each.

1. Write a Python function called `linear_search` that is given two arguments: a value x and a list L. The function must return the list index of the first location of x in L. If x is not in L the function must return -1. You may use any Python `list` functions you wish.

2. What if the list is already sorted? Write a modifed version of `linear_search` that returns the index of the first instance of x or the index where x should be inserted if it is not in L For example, in the list

```
L = [ 1.3, 7.9, 11.2, 15.3, 18.5, 18.9, 19.7 ]
```

   the call

```
linear_search(11.9, L)
```

   should return 3, while the call

```
linear_search( 20.5, L)
```

   should return 7. You must not use binary search (even though that would be faster — this is an exercise) and you must use either a `for` or a `while` loop.

LECTURE 21 — SORTING

39.1 Overview

- Sorting is a fundamental operation

- Provides good practice in implementing and testing small functions

- Leads to a better understanding of algorithm efficiency

- Allows us to consider the fundamental notion of a merge of two sorted sequences.

- During testing, we will see an example of the important notion of passing functions as arguments.

39.2 Algorithms to Study

- Insertion sort

- Merge sort

    - This is our primary focus.

- Python's built in sort

39.3 Insertion Sort

- Idea:

    - If we already have a sorted list and we want to insert a new value, we can shift values one location higher until we find the proper location for the new value

    - Insert the new value

    - Start with a just a list of length 1 and repeat until all values have been inserted

- Algorithm

```
for each index i in the list, starting at 1 do
    Save the value stored at location i in variable x
    Initialize j at location i-1
    while j is non-negative and the location to insert x has not been found do
        Shift the value at location j up to location j+1
        Decrement j
    Insert the value stored in x in location j+1
```

- Code (in-class exercise):

```
def ins_sort(v):
```

39.4 Steps to Testing

1. Re-read and mentally simulate

2. Insert print statements and/or view with debugger to see what it is actually doing.

3. Run on "test cases" that capture challenging conditions:

   - Empty list

   - Singleton list

   - List of repeated values

   - List where the largest value is at the beginning or the smallest value is at the end.

39.5 Rough Analysis of Time Required

- For any particular value of i in the outer for loop, there can be up to i-1 comparisons/shifts.

  - When i==1 this is not much, but

  - When i==N-1, this is a lot.

- Adding across the different values of i, this results in at most (roughly) $N^2/2$ comparisons.

- We write this as $O(N^2)$ because (informally) the number of comparisons done is proportional to N^2.

39.6 Experimental Analysis

- The code lec21_test_sort.py posted on the Piazza site, and attached to the end of these notes, will be used to do timing experiments on all the sorts we write.

- Makes use of the random module

- Includes two main functions:

  - run_and_time

  - generate_local_perm

  We will discuss each of these in turn.

- The sorting functions themselves are functions in the module sort

- Notice that the sorting function is *passed as an argument* to run_and_time:

  - First time that we have passed a function as an argument to another function!

- We will start with experiments to analyze selection sort (see textbook) and insertion sort.

39.7 Breaking the N-Squared Barrier

- The fundamental problem with both selection sort (discussed in the textbook, but not in these notes) and insertion sort:

    - We need to do up to N comparisons by scanning through the list to find the proper location of the next value in the sorted list.

    - For insertion sort, we could use binary search to find the insertion location, but we would still have up to N shifts of values.

- Do better than selection sort and insertion sort by using algorithms that don't scan the entire list to assign one value.

- Examples

    - Quick sort

    - Heap sort

    - Merge sort

- We'll study merge sort, in part because it is the easiest of these to understand and in part because of the importance of the idea of a merge.

39.8 Merging Two Sorted Lists

- Given two lists that are each already sorted, our problem is to generate a new sorted list containing all of the items from both lists.

- For example,

```
L1 = [ 9, 12, 17, 25 ]
L2 = [ 3, 5, 11, 13, 16 ]
```

must be merged into a new list containing

```
[ 3, 5, 9, 11, 12, 13, 16, 17, 25 ]
```

- Idea:

    - Since both lists are sorted, the first item in the new list must be the first item in one of the lists!

    - If we "remove" the smallest item (3 in L1 in this case), the next item will again be the first non-copied item in one of the two lists!

    - We repeat this process until one of the lists has no more items to copy.

    - Then, copy the remainder of the other list to the back of our new list.

- We don't actually remove the items from L1 or L2. Instead we keep an index to the next location of L1 and L2 that has not yet been copied.

- We'll write the code in class, starting from here

```
def merge(L1, L2):
    i1 = 0
    i2 = 0
    L = []
```

```
        return L
```

- Studying the solution:

    1. Write the values of the index variables, `i1` and `i2`, each time through the loop for lists L1 and L2 above.

    2. What are the values of `i1` and `i2` when the loop terminates?

39.9 Merge Sort

- Key observation: all lists of length 1 are sorted

- Therefore, for a list of length N that is to be sorted:

    - Create N lists of length 1 from the values in the list

    - Start to merge these "singleton" lists in pairs to create longer, sorted lists.

    - Repeat on pairs of longer lists in succession

- Requires

    - Keeping a list of sorted sublists, initialized with each singleton list

    - Rather than deleting the sorted sublists, just keep track of which we need to work on.

- Code (in class):

```
def merge_sort(v):
    if len(v) <= 1:
        return
```

39.10 Analysis of Merge Sort

- Check for correctness

- We'll give an informal analysis explaining why there are only $O(N \log N)$ comparisons.

- Experimental timings

- Can you think of ways to improve our implementation of the merge sort idea?

39.11 Final Comparison Across All Sorts

- Selection sort and insertion sort are dramatically slower than merge sort, which in turn is dramatically slower than Python's built-in sort, a highly optimized, C language implementation of merge sort.

- Shows

    - the difference between $O(N^2)$ sorting and $O(NlogN)$ sorting, and

    - the difference between a straight-forward Python implementation and a careful, optimized implementation of the same algorithm.

  Both of these are important!

- Final question: what happens when values are "almost" sorted?

    - Experimentally, we can explore this using the `generate_local_perm` function in `test_sort.py`.

    - Insertion sort becomes much faster, far outstripping selection sort. Why?

39.12 Practice Questions

1. For our insertion sort code, show the contents of the following list after each iteration of the outer `for` loop

   ```
   v = [ 12, 4, 11, 2, 6, 18, 9 ]
   ```

   While you can and should use the implementation to test your answers, you should start by manually generating the answers on your own.

2. Show the contents of the `lists` list at the end of the `merge_sort` implementation developed in class when it is called with

   ```
   v = [ 17, 15, 29, 66, 31, 19, 9, 33 ].
   ```

3. Consider the following function

   ```python
   def extract( comp, v ):
       x = v[0]
       for i in range(1,len(v)):
           if comp(v[i],x):
               x = v[i]
       return x
   ```

   Note that `comp` is a function that has been passed to `extract`.

   (a) Write a function called `compare_lower(a,b)` such that if L is a list then the call

   ```
   extract(compare_lower,L)
   ```

   returns the smallest value in L.

   (b) Write a function called `compare_upper(a,b)` such that if L is a list then the call

   ```
   extract(compare_upper,L)
   ```

   returns the largest value in L.

4. Write a version of `merge` that does all of the work inside the `while` loop and does not use the `extend`. This is a good test of your logic skills.

---

5. Based on your previous solution write a function to merge three sorted lists. This is an even greater challenge to your logic skills.

Note that when it comes to the Final, you will not be required to have memorized the code of the sorting functions, but you should know the algorithms!

Sort Testing Code

```
'''
Testing code for Computer Science 1, Lecture 21 on sorting. This
assumes that the sort functions are all in file lec21_sorts.py, each taking
one list as its only argument, and that their names are sel_sort
ins_sort merge_sort

All tests are based on random permutations of integers.

. In most of our tests, these permutations are completely random,
  meaning that a value is equally likely to end up anywhere in the
  list.

. In the final test we will explore the implications of working
  with lists that are "almost sorted" by only moving values a small
  distance from the correct location.  You can see that insertion sort
  is very fast in this case by removing the # char in front of
  generate_local_perm
'''

import lec21_sorts as sorts
import time
import random


def run_and_time(name, sort_fcn, v, known_v):
    '''
    Run the function passed as sort_fcn, timing its performance and
    double-checking if it correct.  The correctness check is probably
    not necessary.
    '''
    print("Testing " + name)
    t0 = time.time()
    sort_fcn(v)
    t1 = time.time()
    print("Time: {:.4f} seconds".format(t1-t0))
    # print("Is correct?", v==known_v
    print()


def generate_local_perm(v,max_shift):
    '''
    This function modifies a list so values are only a small amount
    out of order.  Each one  Generate a local permutation by randomly moving each
    value up to max_shift locations in the list.
    '''
    for i in range(len(v)):
        min_i = max(0,i-max_shift)
        max_i = min(len(v)-1, i+max_shift)
        new_i = random.randint( min_i, max_i )
        v[i], v[new_i] = v[new_i], v[i]
```

```python
##################################################

if __name__ == '__main__':
    n = int(raw_input("Enter the number of values ==> "))
    print("----------")
    print("Running on {:d} values".format(n))
    print("----------")

    v = range(n)
    v1 = v[:]
    random.shuffle(v1)
    # generate_local_perm(v1, 10)
    v2 = v1[:]
    v3 = v1[:]
    v4 = v1[:]

    run_and_time("merge sort", sorts.merge_sort, v3, v )    # passing functions as an arg to a fcn
    run_and_time("python sort", list.sort, v4, v )
    run_and_time("selection sort", sorts.sel_sort, v1, v )
    run_and_time("insertion sort", sorts.ins_sort, v2, v )
```

LECTURE 21 — EXERCISES

Solutions to the problems below must be sent to Submitty for automatic scoring. A separate file must submitted for each problem. Solutions must be submitted by 4 pm on Tuesday, December 5. Download the file `insert_sort_and_merge.zip` from the course Piazza site.

1. The file `insert_sort.py` contains an implementation of the insertion sort algorithm. It includes `print` function calls. What are the outputs of these statement? Submit a text file showing the output.

2. The file `merge.py` contains an implementation of the `merge` function that is the heart of merge sort. We only consider `merge` here and not the full sort. Modify the merge function so that if the same value appears in both lists then only one copy of the value is in the final merged list. You may assume that each contains no duplicates. Work within the context of the merge function itself, meaning that you should not use a set and you should not use an extra list. You need to only change a few lines of code.

   For example, if the two lists are

   ```
   L1 = [ 2, 7, 9, 12, 17, 18, 22, 25 ]
   L2 = [ 1, 5, 6, 8, 13, 14, 15, 18, 19, 23, 25 ]
   ```

   Then the result `merge(L1,L2)` should be the list

   ```
   [ 1, 2, 5, 6, 7, 8, 9, 12, 13, 14, 15, 17, 18, 19, 22, 23, 25 ]
   ```

   Do not change the name of the function and do not change the name of the file. On Submitty we will run code that imports `merge.py` and calls the function `merge` passing two lists as arguments.

LECTURE 22 — TKINTER

41.1 Last Four Lectures

Selection of advanced topics:

- Lecture 22: Graphical user interfaces
- Lecture 23: Recursion
- Lecture 24: Functional programming
- Lecture 25: Course summary; intro to C++

41.2 Overview of Today's Lecture

- We will learn about event driven programming
- We will discuss graphical user interfaces (GUI)
- We will see how to use TkInter implementation for Python to develop GUIs

41.3 Graphical User Interface

- All our programs so far had a terminal based input/output through Wing. This is the simplest type of user interface: enter some text and get some output.
- In real life, we rarely use such programs. All Web interfaces and mobile apps involve elements like buttons, text input fields, radio buttons.

  This type of a user interface is called a **graphical user interface** or GUI.

- Designing a GUI is not a simple matter. It requires one to understand effective communication of information through visual interfaces. It is highly recommend you take a class on Visual Communication or Human Computer Interaction to learn more about these.

- Many GUI packages will have some default settings that are meant to help you design interfaces that are at least not awful! So, make sure you do not override them until you know what you are doing.

41.4 Event Driven Programming

- A graphical user interface is designed to do something in response to an event.

- Example events are:

    – Button push (most popular so far)

    – Selection from a list of options

    – Mouse movement

- The interface usually runs within a main window (often called root) and listens continuously for events it knows how to process.

    – Hence, a GUI is always in an infinite loop.

    – When an event happens, it executes the function corresponding to the event and continues to listen.

    – To finish the GUI loop, often an explicit termination command has to happen.

- TkInter is a Python module implementing the Tk toolkit for building graphical user interfaces. Other languages also have Tk implementations.

- **Note:** The most important distinction between regular and event based programming is that the action corresponding to an event gets executed in parallel with the rest of the program, it does not block the execution of functions in response to other events.

41.5 TkInter First Program

- Let's see a very basic program.

```
from tkinter import *   ### (1)

root = Tk()            ### (2)
root.mainloop()        ### (3)
print("Hello")         ### (4)
```

1. It is common to import everything from Tkinter to make it easy to write GUI programs. Compare this with the following program that does the same exact thing but requires using Tkinter for all calls to the module.

    ```
    import tkinter
    root = tkinter.Tk()
    root.mainloop()
    print("Hello")
    ```

2. We create the main window that the whole application will run in. This main window has many properties already: it can be resized and moved like any window on your computer.

3. We tell the main window to run in an infinite loop, until it is explicitly terminated. In this case, you need to explicitly click on the upper left corner to close this application. This generates a "destroy" event, which terminates the main window.

4. To understand the infinite loop notice that `print "Hello"` command is not executed until the window terminates and the main loop ends.

41.6 Widgets and Containers

- GUIs involve two main types of objects: widgets and containers

- Widgets are objects that have a function and are visible. These are the main elements of an interface. We will see the following main widgets:

  – Buttons: can do something when clicked

  – Canvas: you can draw shapes or put shape in them

- There are many other widgets like radio boxes, check boxes, menu of items, etc. You can learn these as we get comfortable with these two.

- Containers are invisible, but they allow us to group widgets and arrange them in visual groups. We will see only one type of container in this lecture:

  – Frames: can contain any of the widgets of the above type

- Containers' size will grow and shrink to fit the objects contained within. Containers can contain other containers as well.

41.7 Example use of containers and buttons

- Let's create a first program to create four buttons, put two pairs in one frame, the second pairs in another frame, and then arrange these two frames on top of each other.

```python
from tkinter import *

root = Tk()
main_frame = Frame(root)
main_frame.pack()

top_frame = Frame(main_frame)
top_frame.pack(side=TOP)

bottom_frame = Frame(main_frame)
bottom_frame.pack(side=BOTTOM)

button1 = Button(top_frame, text="Top 1")
button1.pack(side=LEFT)
button2 = Button(top_frame, text="Top 2")
button2.pack(side=RIGHT)

button3 = Button(bottom_frame, text="Bottom 1")
button3.pack(side=LEFT)
button4 = Button(bottom_frame, text="Bottom 2")
button4.pack(side=RIGHT)

root.mainloop()
```

- In the above example, the two frames for containing buttons are organized inside another frame.

- The organization between widgets is achieved by a strict hierarchy. Each widget or container has a parent:

GUI Element	Parent
root	None
main_frame	root
top_frame	main_frame
bottom_frame	main_frame
button1	top_frame
button2	top_frame
button3	bottom_frame
button4	bottom_frame

- Whenever you create a frame, button, canvas, the first argument is the parent element.

  - Parent element tells us the placement of the widget, e.g. `button1` is placed to the left of `top_frame` and `top_frame` is to the top of `main_frame` and `main_frame` is within the main window of `root`.

  - Parent container's size is determined as a function of the sizes of its children. If nothing was put in a container, its size may be zero.

- Pack will make the interface element a part of the GUI, creating them is not enough. You must attach them to the interface with packing.

- By changing the arguments to `pack()` we can alter the placement of the buttons and the frames.

41.8 Running tkinter programs using classes

- Already our programs are becoming long and so far they don't really do anything yet! What we really want is to have programs accomplish tasks when a button is pressed (for example).

- As it is generally tricky to pass variables between different Tkinter events, we often use classes to implement the main Tk applications.

  The member attributes will help us pass values between events.

- We will first rewrite the above program using classes as shown below.

```python
from tkinter import *

class MyApp(object):
    def __init__(self, parent):
        self.main_frame = Frame(parent)
        self.main_frame.pack()

        self.top_frame = Frame(self.main_frame)
        self.top_frame.pack(side=TOP)
        self.bottom_frame = Frame(self.main_frame)
        self.bottom_frame.pack(side=BOTTOM)

        self.button1 = Button(self.top_frame, text="Top 1")
        self.button1.pack(side=LEFT)
        self.button2 = Button(self.top_frame, text="Top 2")
        self.button2.pack(side=RIGHT)
        self.button3 = Button(self.bottom_frame, text="Bottom 1")
        self.button3.pack(side=LEFT)
        self.button4 = Button(self.bottom_frame, text="Bottom 2")
        self.button4.pack(side=RIGHT)
```

```
if __name__ == "__main__":
    root = Tk()
    myapp = MyApp(root)
    root.mainloop()
```

- This is the main idea: Any object outside of root is defined within a class, using root as the parent.

- We can now add functionality to this class that will modify the widgets.

41.9 Button Click

- You track many events in a GUI, but we will not see them (such as when a specific mouse button, left or right, is clicked, when it is released, when the cursor is moved, when something is typed on the keyboard, etc.)

  (Note: The distinction between the widget that is the GUI button, and the physical button on your mouse.)

- Command attribute of *Button* allows you to associate a function name with a button click (in GUI terms this is a combination of two events: left button click and its release).

  See the following example:

```
self.button = Button(self.bottom_frame, text="Quit", command=self.terminate_program)
```

- The button we generate in this example has a function binding: when it is clicked, it will call the function called *self.terminate_program*.

41.10 Example Program with a Button Click

- This simple program terminates the program by completely destroying the root window.

```
from tkinter import *

class MyApp(object):
    def __init__(self, parent):
        self.parent = parent
        self.main_frame = Frame(parent)
        self.main_frame.pack()
        self.button = Button(self.main_frame, text="Quit", command=self.terminate_program)
        self.button.configure(width=12, padx="4m", pady="4m")
        self.button.pack()

    def terminate_program(self):
        self.parent.destroy()

if __name__ == "__main__":
    root = Tk()
    myapp = MyApp(root)
    root.mainloop()
```

- Here we see the use of classes for two important functions:

  – Keeping track of the root as the parent of all the GUI elements.

  – Defining a function that can access the parent object attached to *self* and destroy it by ending the application.

- We also saw that you can control the size of the button by configuring its width. Padding on x and y allows us to control how the button is placed inside the frame.

41.11 Canvas Widget

- This is the last widget we will see today. Canvas is a blank area. You can continuously draw things in it or put text in it. Here are some basic operations.

- Create canvas:

```
canvas = Canvas(parent, height=200, width=200)
canvas.pack()
```

- Draw something in a canvas (an oval within the given box coordinates):

```
canvas.create_oval((40,40,80,80))
```

- Remember, like Image objects, (0,0) represents the top left corner.

- When you draw multiple things in a canvas, it may not immediately show what you draw and a small delay may be noticed. You can force the canvas to show you what you draw with:

```
canvas.update()
```

- You can also draw lines, rectangles and text. See more on:

  http://effbot.org/tkinterbook/canvas.htm

41.12 Timing of GUI Events

- Remember, GUI events are triggered when you click a button (or other events you watch in more sophisticated systems).

- They execute in parallel. If the processing of an event takes some time, it may overlap with another event.

- Sometimes drawing events are too fast for the eye to see the progress. You can incorporate some delays to make this more pleasing to the eye.

```
canvas.after(waittime)
```

  which will add a small delay *waittime* to the program.

- Note that *after* and *update* functions apply to many other GUI elements. Mastering them will help you get the exact timing effect from your interface.

41.13 Final Program

- Here is a final program to put together a lot of the things we learned. It features a private class method (*new_button*) that is only accessible to the *__init__* function. It shows how functions can be used to automate repetitive tasks.

```python
from tkinter import *

class MyApp(object):
    def __init__(self, parent):
        ## This method is internal to the initializer method
        ## and is used for creating buttons. It shortens the program code
        def new_button(parent, cmd, buttontext, packlocation):
            button = Button(parent, command=cmd)
            button.configure(text=buttontext)
            button.configure(width=button_width,
                    padx=button_padx, pady=button_pady )
            button.pack(side=packlocation)
            return button

        #------ constants for controlling layout ------
        button_width = 10
        button_padx = "2m"
        button_pady = "1m"
        buttons_frame_padx =  "3m"
        buttons_frame_pady =  "2m"
        buttons_frame_ipadx = "3m"
        buttons_frame_ipady = "1m"
        # -------------- end constants ----------------

        #---------variables for controlling the function-----
        self.canvas_dimension = 600 ##Canvas will be a square
        self.wait_time = 8
        self.repetitions = 2
        #----------end of variables-------------------------

        self.myParent = parent
        self.main_frame = Frame(parent)
        self.main_frame.pack ()

        ## Two frames inside the main frame, one for the canvas
        ## on top and the second one for buttons in the bottom
        self.draw_frame = Frame(self.main_frame)
        self.draw_frame.pack(side=TOP)

        self.info_canvas = Canvas(self.draw_frame, height=20,
                                  width=self.canvas_dimension)
        self.info_canvas.pack(side=TOP)
        self.text_area = self.info_canvas.create_text(10,10,anchor='nw')
        self.info_canvas.itemconfigure(self.text_area,text="#circles = {:d}".format(self.
        ↪repetitions))

        self.main_canvas = Canvas(self.draw_frame, \
                                  height=self.canvas_dimension,
                                  width=self.canvas_dimension)
        self.main_canvas.pack()

        self.button_frame = Frame(self.main_frame)
        self.button_frame.pack(side=BOTTOM)

        self.draw_button = new_button(self.button_frame,self.draw, 'Draw', LEFT)
        self.clear_button = new_button(self.button_frame,self.clear, 'Clear', LEFT)
        self.increase_button = new_button(self.button_frame,self.increase, 'Increase', LEFT)
```

41.13. Final Program

```
        self.reduce_button = new_button(self.button_frame,self.reduce, 'Reduce', LEFT)
        self.quit_button = new_button(self.button_frame,self.quit, 'Quit', RIGHT)

    def clear(self):
        self.main_canvas.delete("all")

    def reduce(self):
        if self.repetitions > 1:
            self.repetitions //= 2
        self.put_info()

    def increase(self):
        if self.repetitions < 200:
            self.repetitions *= 2
        self.put_info()

    def put_info(self):
        ## Change the text field in the canvas
        self.info_canvas.itemconfigure(self.text_area,text="#circles = {:d}".format(self.
→repetitions))

    def draw(self):
        boundary_offset = 2
        max_radius = (self.canvas_dimension - 2*boundary_offset) // 2
        xc = self.canvas_dimension//2 + boundary_offset
        r = max_radius/self.repetitions
        inc = r
        for i in range(self.repetitions):
            self.main_canvas.create_oval((xc-r, xc-r, xc+r, xc+r))
            r += inc
            self.main_canvas.update() # Actually refresh the drawing on the canvas.
            # Pause execution.  This allows the eye to catch up
            self.main_canvas.after(self.wait_time)

    def quit(self):
        self.myParent.destroy()

if __name__ == "__main__":
    root = Tk()
    root.title("Drawing a circle") ##Give a title to the program
    myapp = MyApp(root)
    root.mainloop()
```

41.14 Summary

- Graphical user interfaces are event driven. You need to write functions to initialize the interface and change the interface when events happen.

- The most common events are button clicks, but many others are possible.

- Most GUI elements are visible and are called widgets.

- Widgets are placed in invisible containers like frames to group them together.

- Many containers have built-in methods for placing multiple widgets in a way that is pleasing to the eye.

- When building an interface, some of the challenges involve making the interface easy and intuitive to use. Many communications classes concentrate on these issues.

LECTURE 22 — EXERCISES

Solutions to the problems below must be sent to Submitty for automatic scoring. A separate file must submitted for each problem. Solutions must be submitted by 4 pm on Friday, December 1.

1. What is printed out when the following code is run? Assume that the user clicks on the "Launch" button and nothing else. Put your answer into a text file and submit it as the answer to part 1.

```python
from tkinter import *

class MyApp(object):
    def __init__(self, parent):
        print('Initialized!')
        self.parent = parent
        self.main_frame = Frame(parent)
        self.main_frame.pack()
        self.button1 = Button(self.main_frame, text="Launch", command=self.terminate_program)
        self.button1.configure(width=12, padx="4m", pady="4m")
        self.button1.pack()
        self.button2 = Button(self.main_frame, text="C", command=self.c)
        self.button2.configure(width=12, padx="4m", pady="4m")
        self.button2.pack()
        self.rocket = 2

    def d(self):
        self.rocket += 2

    def c(self):
        print("Launching ... counting", end='')
        for i in range(self.rocket):
            print('...{}'.format(self.rocket), end='')
            self.rocket -= 1
        print('...Liftoff!', flush=True)

    def terminate_program(self):
        self.c()
        print('Terminated')
        self.parent.destroy()

if __name__ == "__main__":
    root = Tk()
    print("Countdown ready")
    myapp = MyApp(root)
    myapp.d()
    root.mainloop()
    myapp.d()
```

```
    print(myapp.rocket)
    myapp.c()
    print("Done")
```

2. Revisiting the same code, what is printed if the code is run a second time and the user clicks the "C" button and nothing else? Put your answer into a text file and submit it as the answer to part 2.

LECTURE 23 — RECURSION

43.1 Overview

- When a function calls itself, it is known as a recursive function.

- Use of the function call stack allows Python to handle recursive functions correctly.

- Examples include factorial, Fibonacci, greatest common divisor, flattening a list of lists, and mergesort.

- We'll think about how to hand-simulate a recursive function as well as rules for writing recursive functions.

43.2 Our First Example

- Consider the following Python function.

```python
def blast(n):
    if n > 0:
        print(n)
        blast(n-1)
    else:
        print("Blast off!")
```

- What is the the output from the call?

```python
blast(5)
```

43.3 Python's Call Stack Mechanism

The following mechanism helps us understand what is happening:

- Each time the code makes a function call, Python puts information on the "call stack", including

  - All values of parameters and local variables

  - The location in the code where the function call is being made.

- Python then makes the function call, switching execution to the start of the called function.

- This function in turn can make additional (potentially recursive) function calls, adding information to the top of the stack each time.

- When a function ends, Python looks at the top of the stack, and

- restores the values of the local variables and parameters of the most recent calling function,

- removes this information from the top of the stack,

- inserts the returned value of the called function (if any) in the appropriate location of the calling function's code, and

- continues execution from the location where the call was made.

43.4 Practice Problems to Illustrate This

What are the outputs of the following?

```python
def rp1( L, i ):
    if i < len(L):
        print(L[i], end=' ')
        rp1( L, i+1 )
    else:
        print()

def rp2( L, i ):
    if i < len(L):
        rp2( L, i+1 )
        print(L[i], end=' ')
    else:
        print()

L = [ 2, 3, 5, 7, 11 ]
rp1(L,0)
rp2(L,0)
```

Note that the entirety of list L is not copied to the top of the stack. Instead, a reference (an alias) to L is placed on the stack.

43.5 Factorial

- The factorial function is

$$n! = n(n-1)(n-2)\cdots 1$$

and

$$0! = 1$$

- This is an imprecise definition because the ... is not formally defined.

- Writing this recursively helps to clear it up:

$$n! = n \cdot (n-1)!$$

and

$$0! = 1$$

The factorial is now defined in terms of itself, but on a smaller number!

- Note how this definition now has a recursive part and a non-recursive part:
  - The non-recursive part is called the *base case.* There must be *at least* one base case in every recursive function definition.

43.6 Exploring Factorial

We will:

- Write a recursive Python function to implement $n!$.
- Hand-simulate the call stack for $n = 4$.

We'll add output code to the implementation to help visualize the recursive calls in a different way.

43.7 Guidelines for Writing Recursive Functions

1. Define the problem you are trying to solve in terms of smaller / simpler instances of the problem. This includes
   (a) What needs to happen before making a recursive call?
   (b) What recursive call(s) must be made?
   (c) What needs to happen to combine or generate results after the recursive call (or calls) ends?
2. Define the base case or cases.
3. Make sure the code is proceeding toward the base case in every step.

43.8 Fibonacci

- The Fibonacci sequence starts with the values 0 and 1.
- Each new value in the sequence is obtained by adding the two previous values, producing

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$$

- Recursively, the nth value, f_n, of the sequence is defined as

$$f_n = f_{n-1} + f_{n-2}$$

- This leads naturally to a recursive function. . .

43.9 Dangers of Recursion

- Some recursive function implementations contain wasteful repeated computation.
- Recursive function calls — like any function calls — typically involve hidden overhead costs.
- Often, therefore, a recursive function can (and should) be replaced with a non-recursive, *iterative* function that is significantly more efficient.
- This is easy to do for both Factorial and Fibonacci, as we will see in class.

43.10 Why, Then, Do We Study Recursion?

- Many of our definitions and even, our logical structures (such as lists), are formalized using recursion.

- Sometimes recursive functions are the first ones we come up with and the easiest to write (at least after you are comfortable with recursion).

  – Only later do we write non-recursive versions.

- Sometimes on harder problems it is difficult to even write non-recursive functions! The list flattening problem below is one such example.

43.11 Advanced Examples

We'll spend the rest of class on three more advanced examples:

1. Recursive geometric shapes: the Sierpinski triangle

2. Flattening a nested list

3. A recursive version of merge sort

43.12 Recursive Geometric Shapes

- Fractals are often defined using recursion. How do we draw a Sierpinski triangle like the one shown below?

- We will look at this example in class and attempt to define the recursion.

- To aid us, we'll look at a Tkinter Python program that implements the drawing of the Sierpinski triangle.

43.13 Flattening a Nested List

- Suppose we want to take a list such as

```
v = [[1,5], 6, [[2]], [3, [7, 8, [9,10], [11,12] ]]]
```

and "flatten" it, converting it to just a list of values with no sublists.

```
v = [ 1, 5, 6, 2, 3, 7, 8, 9, 10, 11, 12 ]
```

- This is challenging because we don't know when we write a function to solve this problem how "deep" the nesting of sublists goes. The solution should handle arbitrary depths:

  - Many, many data structures (containers) in computer science have this type of nested / recursive structure: an entry in a list may be another list. . . .

- To solve this problem we will also need to know how to recognize when an entry in a list is another list. For this we need to use the `type` function in Python. The following example should make this clear:

```
>>> v = [ 'a', 'b', 'c']
>>> x = 12
>>> type(v) == list
True
>>> type(v[0]) == list
False
>>> type(x) == int
True
```

- Now we are ready to solve the "flattening" problem. We'll look at two different approaches. In both, the key will be to distinguish between handling elements that are lists (and therefore must be flattened recursively) and elements that are not lists. We'll start from. . .

```
def flatten(L):




if __name__ == "__main__":
    v = [[1,5], 6, [[2]], [3, [7, 8, [9,10], [11,12] ]]]
    print(v)
    print(flatten(v))
```

43.14 Final Example: Merge Sort

- The fundamental idea of merge sort is recursive:

  - Any list of length 1 is sorted

  - Otherwise:

    * Split the list in half

    * Recursively sort each half

        ∗ Merge the resulting sorted halves

- We repeat our use of the `merge` function from Lecture 20:

```python
def merge(L1,L2):
    i1 = 0
    i2 = 0
    L = []
    while i1<len(L1) and i2<len(L2):
        if L1[i1] < L2[i2]:
            L.append(L1[i1])
            i1 += 1
        else:
            L.append(L2[i2])
            i2 += 1
    L.extend(L1[i1:])
    L.extend(L2[i2:])
    return L
```

- Using this, we will write the main `merge_sort` function in class.

```python
def merge_sort(L):
```

- – The solution will be posted on-line.

- – We will also see how a minor mistake can cause the recursion to go into an infinite loop.

- Comparing what we write to our earlier non-recursive version of merge sort shows that the primary job of the recursion is to organize the merging process!

43.15 Summary

- Functions that call themselves are known as "recursive functions"

- Use of a function call stack allows Python to handle recursive functions correctly.

- Many structures and functions important to computer science are defined recursively.

- Fundamentally, recurision is about defining a problem solution as a function of the solution to a simpler/shorter/smaller version of the problem.

- A basis case (or cases) is (are) always needed to make a recursion function succeed.

- Infinite recursion is avoided by ensuring that progress is made toward the basis case or cases in every recursive call.

- While many recursive functions are easily rewritten to remove the recursion, some advanced problems are difficult to solve without recursion.

43.16 Additional Practice Exercises

1. Euclid's algorithm for finding the greatest common divisor is one of the oldest known algorithms. If a and b are positive integers, with $a \geq b$, then let r be the remainder of dividing a by b. If $r == 0$, then b is the GCD of the two integers. Otherwise, the GCD of a and b equals the GCD of b and r. Here is the Python code:

```
def gcd(a,b):
    if a < b:
        a,b = b,a

    r = a % b
    if r==0:
        return b
    else:
        return gcd(b,r)
```

(a) What is the output of

```
print(gcd(36,24))
print(gcd(84,65))
print(gcd(84,66))
```

(b) Why do we know that gcd is proceeding toward the base case (as required by our "rules" of writing recursive functions)?

2. Specify the recursive calls and return values from our merge_sort implementation for the list

```
L = [ 15, 81, 32, 16, 8, 91, 12 ]
```

LECTURE 23 — EXERCISES

Solutions to the problems below must be sent to Submitty for automatic scoring. A separate file must submitted for each problem. Start by downloading *lec23_ex.zip* from the Piazza site. Solutions must be submitted by 4 pm on Tuesday, April 24.

1. The following code recursively calculates the maximum value in a list.

```python
def recursive_max_impl( L, i ):
    '''
    The actual recursive function.
    '''
    if i == len(L)-1:
        return L[i]
    else:
        return max(L[i], recursive_max_impl(L,i+1) )

def recursive_max( L ):
    '''
    The driver for the recursive function.  This handles the special
    case of an empty list and otherwise makes the initial call to the
    recursive function.
    '''
    if len(L) == 0:
        return -99999    # By convention
    else:
        return recursive_max_impl( L, 0 )

if __name__ == "__main__":
    L1 = [ 5 ]
    print(recursive_max( L1 ))
    L2 = [ 24, 23.1, 12, 15, 1 ]
    print(recursive_max( L2))
    L2.append( 55 )
    print(recursive_max( L2 ))
```

   Using this as a guide, write a recursive function to add the values in a list. You should have to change very little code. Implement your code in the provided file prob1.py

2. Implement a recursive solution to the Fibonacci number function definition given during lecture. Implement your code in the provided file prob2.py.

3. Looking carefully at the Fibonacci definition shows that calculating Fibonacci number f_{n-1} requires calculating Fibonacci number f_{n-2}, which is also required for calculating Fibonacci number f_n. This means there is redundant computation. This redundancy gets worse for numbers f_{n-3}, f_{n-4}, etc.

   Fortunately, the Fibonacci sequence is relatively easy to compute non-recursively. That is your problem here.

The trick is to build up the solution using a for loop that calculates f_2, then f_3, then f_4, etc. Implement your solution in `prob3.py`.

Out of curiosity, you could run your solutions to the previous two problems on large values of n and time the difference.

FORTYFIVE

LECTURE 24 — ADVANCED PYTHON TOPICS AND FUNCTIONAL PROGRAMMING

45.1 Problems We'd Like to Solve

Some of these are toy problems, but they illustrate use of tools we'd like to develop and use:

1. How many values are in a list of lists?

2. What is the maximum distance from the origin of the points in a list?

3. What is the sum of squares of the first n integers?

4. Can you sum the positive values in a list?

5. Can you sort a list of points by y value (2nd coordinate) and then by x value?

45.2 Solution Techniques

- We can solve most of these with a `for` loop, but they can be solved even more effectively / efficiently / compactly using advanced Python methods.

- Leads to notions of:

- map and filter

- functions as parameters

- lambda functions

- stable sort

- list comprehensions

- Most are examples of *functional programmming*

45.3 Map: Apply a function to each element of a list

- Suppose we want to count the number of values in a list of lists. We can use `map` to apply the `len` function to each sublist.

```
>>> v = [ [2, 3, 5, 7], [11,13,17,19], [23, 29], [31,37] ]
>>> print( list(map( len, v)) )
[4, 4, 2, 2]
```

- `map` is an *iterator* class:
  - It produces values in a sequence, one after another, by applying the function (1st argument) to the values of the second argument.
  - Technically, an iterator class is one that has the `__next__` method implemented (correctly).
  - Using `list` gives us the list of lengths of the sublists explicitly.
- To complete the solution we need to just apply sum:

```
>>> print(sum(map(len,v)))
12
```

Notice that this does not explicitly form an intermediate list.

45.4 Passing Functions as Parameters

- The above example passes the `len` function as an argument!
  - We also passed functions as arguments to our callbacks in our GUI programs
- This illustrates the concept that Python treats function as "first-class" objects - in other words functions can be used just like variables and other data.
  - What's passed as an argument to `map()` is the location of the function code.
- Now suppose we want to find the maximum distance of a list of points from the origin. Here we'll have to write a function

```python
def dist2D( p ):
    return (p[0]**2 + p[1]**2)**0.5

pts = [ (4.5, 3), (2.1,-1), (6.8,-3), (1.4, 2.9) ]
print(max( map(dist2D,pts) ))
```

45.5 Lambda functions: Anonymous functions

- We can avoid the need to write a separate function here by writing an *anonymous* function called a *lambda* function.
- Our first example is just squaring the values of a list

```
>>> list(map( lambda x: x**2, [ 1, 2, 3, 4 ] ))
[ 1, 4, 9, 16 ]
```

- Now, we can sum the squares from 1 to n

```
>>> n = 100
>>> sum( map( lambda x: x**2, range(1,n+1)))
```

- We can also implement the `dist2D` function anonymously:

```
>>> max( map( lambda p: (p[0]**2 + p[1]**2)**0.5, pts) )
7.432361670424818
```

- Notice that we did not need to explicitly form a list in each of the preceeding examples. This leads to sub-stantial savings when the list is large!

- Aside: the notion of a lambda function goes all the way back to the origin of computer science

45.6 In-Class Practice Problem:

1. Starting with the following list of x,y point coordinate types, we will use `map()`, a lambda function, and `max()` to find the maximum x coordinate (the 0-th coordinate) in a list of points.

```
pts = [ (6,-1), (8,4), (7.5,-3), (4.4,12), (7,2) ]
```

45.7 Filter: Extract / eliminate values from a list

- Consider a different problem: how to eliminate all of the negative values from a list. Based on what we know so far, this requires a for loop with append.

- We can simplify this using the built-in Python construct called `filter`

```
>>> v = [ 1, 9, -4, -8, 10, -3 ]
>>> list(filter( lambda x: x>0, v))
[1, 9, 10]
```

- Here,
   - The lambda function must produce a boolean value and if that value is `True` the list item is kept; oth-erwise it is eliminated.
   - The result of `filter` is an iterator object, just like the result of `map` is. We convert to a list in order to display the answer.

- If we want to sum up the non-negative values, then we don't need to explicitly generate a list:

```
>>> sum(filter( lambda x: x>0, v))
20
```

45.8 Lecture Exercises, Problems 1 and 2:

- At this point students will be given the chance to work on the first two lecture exercises.

45.9 Passing Functions to Sort

- Consider the problem of sorting a list of (x,y) points by their y values first and their x values for tied y values, both in decreasing order. For example, given

```
pts = [ (2,5), (12,3), (12,1), (6,5), (14, 10), (12, 10), \
          (8,12), (5,3) ]
```

we'd like the sorted order to be

```
[(8, 12), (14, 10), (12, 10), (6, 5), (2, 5), (12, 3), \
    (5, 3), (12, 1)]
```

- The Python sort function

```
>>>  sorted( pts, reverse=True )
[(14, 10), (12, 10), (12, 3), (12, 1), (8, 12), (6, 5), \
    (5, 3), (2, 5)]
```

gives the ordering by x value and then by y value. This is not what we want.

- The first step to a solution is to provide a *key* function to `sorted()` to pull out the information (the y value in this case) from each tuple to use as the basis for sorting:

```
>>> sorted( pts, key = lambda p: p[1], reverse=True)
[(8, 12), (14, 10), (12, 10), (2, 5), (6, 5), (12, 3), \
    (5, 3), (12, 1)]
```

This is close but not quite right because the two points with y=5 are out of order.

- The trick is to sort by x first and then sort by y!

```
>>> by_x = sorted(pts,reverse=True)
>>> by_x
[(14, 10), (12, 10), (12, 3), (12, 1), (8, 12), (6, 5), \
   (5, 3), (2, 5)]
>>> sorted( by_x, key = lambda p: p[1], reverse=True)
[(8, 12), (14, 10), (12, 10), (6, 5), (2, 5), (12, 3), \
   (5, 3), (12, 1)]
```

- This works because `sorted()` uses what's known as a *stable sort*: when two values are "tied" according the sorting criteria (y value in the second sort) their relative ordering (by x value from the first sort) in the final list is preserved.

  – Therefore, (6,5) comes earlier than (2,5), while (12,3) comes earlier than (5,3)

- A number of variations on sorting use this "stable sort" property, but not all fast sorting algorithms are stable.

- Of course, we can also extend our lambda to reverse the tuple provided to sort()

```
>> sorted( pts, key = lambda p: (p[1], p[0]), reverse=True)
[(8, 12), (14, 10), (12, 10), (6, 5), (2, 5), (12, 3), \
   (5, 3), (12, 1)]
```

45.10 Practice Problem

1. Use `filter` to eliminate all words that are shorter than 4 letters from a list of words

45.11 List Comprehensions

- Instead of `map` and `filter` some people prefer another example of functional programming in Python called *list comprehensions*

- Here is an example to generate a list of the squares of the first n integers:

```
>>> n = 8
>>> [ i*i for i in range(1,n+1) ]
[1, 4, 9, 16, 25, 36, 49, 64]
```

- The form of this is an expression followed by a `for` loop statement.

- We can get the effect of `filter` by adding a conditional at the end:

```
>>> v = [ 1, 9, -4, -8, 10, -3 ]
>>> [ x for x in v if x>0 ]
[1, 9, 10]
```

- Here, the values are only generated in the resultant list when the `if` condition passes.

- We can combine these as well. As a slightly silly example, we can eliminate the negative values and square the positive values

```
>>> v = [ 1, 9, -4, -8, 10, -3 ]
>>> [ x*x for x in v if x>0 ]
[1, 81, 100]
```

- We can get even more sophisticated by nesting `for` loops. Here is an example where we generate all pairs of numbers between 1 and 4, except for the pairs where the numbers are equal

```
>>> [ (i,j) for i in range(1,5) for j in range(1,5) if i != j ]
[(1, 2), (1, 3), (1, 4), (2, 1), (2, 3), (2, 4), (3, 1), (3, 2),
    (3, 4), (4, 1), (4, 2), (4, 3)]
```

45.12 Exercises

1. Write a list comprehension statement to generate a list of all pairs of odd positive integer values less than 10 where the first value is less than the second value.

45.13 Summary and Discussion

- We've explored programming that is more compact and at a higher level of abstraction. It can be used to effectively interact with data.

- `map` and `filter` each take a function and a sequence (an "iterable") as arguments and produce an iterator as a result:

    - `map` produces the result of applying the function to each element of the iterable

    - `filter` produces each element of the iterable for which the function returns `True`

- Both `map` and `filter` are made more compact by using `lambda` functions

- `lambda` functions can also be used to change the result of sorting

- A *stable sort* preserves the relative order of "tied" values

- *List comprehensions* can be used in place of `map` and `filter`:

    - Some people prefer list comprehensions because they often do not require lambda functions, but...

    - List comprehensions explicitly construct the list of results rather than generating them one-by-one, which is what `map` and `filter` do. This makes them less efficient for large data sets.

- These are all examples of *functional programming*.

- We've also used the other major programming paradigms this semester

  - *imperative programming*

  - *object oriented programming*

- Many modern languages like Python provide tools that allow programming using a combination of paradigms

FORTYSIX

LECTURE 24 — EXERCISES

Solutions to the problems below must be sent to Submitty for automatic scoring. A separate file must submitted for each problem. Start with the files available on the Piazza resource page as *lec24_ex_files.zip*. Solutions must be submitted by 4 pm on Friday, April 27. **Each problem requires that you write / modify exactly one line of code!**

1. Modify the code in `prob1.py` to generate a new list where all values are replaced by their absolute values. You must use `map`, but there is no need for a lambda function.

2. Modify the code in `prob2.py` to determine the minimum x value of all points that are in the first quadrant (x and y values both positive). This will require you to use `map`, `filter` and `min`. You can write this in a single line, but for clarity you are welcome to write it on two or three. Do not create a new list.

3. Modify the code in `prob3.py` to sort the points in a list by their distance from the origin (i.e. by the magnitude of the points). This will require use of a `lambda` function.

4. Modify the code in `prob4.py` to use a list comprehension statement to convert a list of Fahrenheit temperatures to Celsius, while eliminating those that are below freezing.

Congratulations: You are finished with the CS 1 lecture exercises!!

TRANSITION TO THE NEXT CLASS - DATA STRUCTURES

47.1 Overview

If you are continuing on your journey in Computer Science, here are some suggestions for you.

- The next class in the sequence will introduce you to completely new concepts, building on the concepts learnt in this class.

- Make sure you are very comfortable with the concepts we learnt here. Review labs and homeworks, make sure you can solve them pretty easily.

- Install a working environment in C++, which is the language you will be using in the next class.

- Review C++ syntax before you take your next class. Make sure you develop:

    - Skills to solve the CS-1 level problems in C++.

    - Skills to debug C++ programs.

    Students who wrote basic C++ programs over the break did well in Data Structures. It also helps solidify your algorithmic problem solving skills.

- You can also review the first few labs in Data Structures if you wish and work on those in advance:

    http://www.cs.rpi.edu/academics/courses/fall17/csci1200/calendar.php

In this document, there are some suggestions to help you get ready for your next class.

47.2 Learning C++

You are not learning C++ from scratch. Many basic constructs are very similar in C++, but there are differences too. Here are some sources that were highly recommended:

- Tutorial on C++:

    http://www.cplusplus.com/doc/tutorial/

- See also this for a quick overview of syntax:

    http://www.cs.rpi.edu/academics/courses/fall17/csci1200/other_information.php

- Another good site for video tutorials:

    https://www.thenewboston.com/index.php

Go through the tutorial and the videos. Make sure to write many programs.

47.3 Differences between Python and C++

Here are a few first things to notice between the two:

- C++ is a compiled statically typed language. This means that variables need to be defined with a data type first, and the data type cannot be changed.

- Programs cannot be run until compiled. If you have a syntax error, the compiler will not produce an executable until you resolve all syntax errors.

- In C++, many new errors will be checked at compile time, like trying to add a string and an integer. As Python is dynamically typed, you are used to resolving such errors at run time. This means, you will spend more of your time in fixing compilation errors in C++ than in Python. Hence, learn to write programs in small steps, test and expand.

- Syntax differences: each line in a program must end with a semicolon and each block must be delimited by curly brackets.

- Be aware of the scope of local/global variables. Generally using our convention of passing variables to functions only as arguments and returning the result will help you a lot.

- Strings and a single character are the same type of object in Python, but not in C++. Overall, make sure you understand each data type well first. Start with simple data types before you go into containers.

- Many basic programming constructs are similar: variables, data types, operations, operator precedence, functions, loops, files, etc. You will find that for loops generally involve a single integer counter, no range function. Learn the basic snytax for these first.

- Containers will be different. Instead of lists in Python that can be used for everything, you will need to learn about arrays, lists and vectors in C++. All containers we used had flexible length, while in C++, some will have predefined size.

- You will see classes in C++ are very similar, but have a few new limitations like private attributes that can only be used within the class. This is actually a good thing, believe me.

For more in depth description of the differences see:

> http://cs.slu.edu/~goldwasser/publications/python2cpp.pdf

> http://www4.wittenberg.edu/academics/mathcomp/shelburne/comp255/notes/Python2Cpp.pdf

47.4 Setting up your C++ Working Environment

Your basic development environment in C++ consists of an editor for writing programs, a Terminal window for compiling and running your programs and a C++ compiler that you must install.

Here are the detailed install instructions.

- You must have a C++ compiler installed.

- You must have a terminal to run the compiler on. The terminal replaces the "Python Shell" version of Wing for you.

- You must use an editor to write your programs. This replaces the editor portion of Wing. Many choices exist. The choice of editor is not so important, but becoming proficient in using an editor is.

As most of the installation will generally involve getting familiar with an editor, here are some common terminal operations below.

47.5 Basic Terminal Operations

Basic terminal operations (Terminal in Macs and Cygwin in PCs) include:

- Displaying the current working directory the terminal is pointing to:

```
$ pwd
```

- Listing the files in the current working directory for the terminal:

```
$ ls
$ ls p*   ##lists files that start with p
$ ls Dropbox   ## lists files in the directory Dropbox
               ## (assuming it is one level below it)
```

- Changing the working directory to a directory on your file system (cd means change directory):

```
$ cd directory_name
$ cd directory1/directory2   ## if you want to change multiple
                             ## levels of directory in a single step
$ cd ..  ## goes one level up in the dictionary hierarchy
```

- Compile code:

```
$ g++ programname -o outputname
```

For example:

```
$ g++ program.cc -o program
```

This generates a new file called *program* in Macs and *program.exe* in PCs.

- To run a program called *program*, use:

```
$ ./program
```

- Create a symbolic link (a shortcut) between two folders is very good way to simplify your life. Suppose you are at your own Desktop. I will create a shortcut to a directory on my dropbox on my own Desktop:

```
$ pwd
/Users/sibeladali/Desktop
$ ln -s /Users/sibeladali/Dropbox/CSCI1100_Fall2014/ cs1
```

This tells me to create a symbolic link (*ln -s*) for directory */Users/sibeladali/Dropbox/CSCI1100_Fall2014/* and call it *cs1*. Now, I can simply write the following:

```
$ cd cs1
```

You are now ready to install your C++ setup.

Here are the detailed install instructions.

SETTING UP C++ FOR DATA STRUCTURES

This material is compiled with the help of many mentors over the years. Special thanks to Micheal Gardner and Lee Cattarin for their edits on prior versions of this document.

Remember to leave install fest with the following tools and skills:

- A compiler that works

- Ability to compile and run a program on the terminal

It is also a good idea to have an editor or an IDE that works. If you have additional time, install an editor and ask a mentor or TA to show you how to use it. But this is less important. Below are instructions on possible ways to accomplish this.

48.1 Installing a terminal and a compiler

Data Structures requires that your C++ code compile and run on the `gcc/g++ 4.8.x` compiler. You may also use `clang++`, which provides more concise and clear error messages. How you install a compiler will depend on your operating system, which requires the compiler program and a Terminal to run it on.

- If you have *Windows 10,* you will likely want to install Windows Subsystem for Linux (WSL). Detailed instructions to install this can be found on the Data Structures course website:

  http://www.cs.rpi.edu/academics/courses/fall17/csci1200/wsl.php

  BUT you only need to complete steps 1 through 7 for now.

- If you have **Windows 7 or 8** WSL is not available and your best option is installing `Cygwin`. Detailed instructions to install this can be found on the Data Structures course website:

  http://www.cs.rpi.edu/academics/courses/fall17/csci1200/cygwin.php

  BUT it is not necessary to go through "Helpful edits to the Cygwin `.bashrc` file" right now.

- If you have a **Mac**, you'll need **XCode**, which provides both a compiler and an IDE for you.

  For Data Structures, **make sure to install XCode's "Command Line Tools"** so that you can compile and run your code in the terminal as well as the IDE. Install instructions for XCode and XCode Command Line Tools can be found:

  http://railsapps.github.io/xcode-command-line-tools.html

  You can then run a Terminal window: Terminal under Applications->Utilities in Macs (black box)

- If you have a Linux OS such as **Ubuntu**, installing is easy. Run the following three commands in a terminal window:

```
sudo apt-get install g++
sudo apt-get install clang++-3.8
sudo apt-get install gcc-multilib g++-multilib
```

48.2 Choosing an editor or IDE (Integrated Development Environment)

A text editor will allow you to view and edit your code (often with syntax highlighting or other features), but not compile and run - you have to do that in a terminal (cygwin, for Windows). An IDE will do all of these things, but might take more time to learn how to navigate as they are usually fuller-featured. Whichever you choose, **you will still have to learn how to compile and run your code in a terminal**, as that is the primary method to get help or show completeness in lab.

- If you want a text editor, these options are popular:
  - Sublime Text: http://www.sublimetext.com/
  - Atom (very similar to Sublime Text): https://atom.io/
  - Notepad++: https://notepad-plus-plus.org/ Windows only, note that this is **not** Notepad
  - Gedit: https://wiki.gnome.org/Apps/Gedit#Download (this is usually installed by default on some Linux systems)
  - There are also terminal editors like

    Vim: http://www.vim.org/

    Nano: https://www.nano-editor.org/

    Emacs: https://www.gnu.org/software/emacs/

    but these may have a steeper learning curve!

- If you want an IDE, these options work with the compiler you'll be using:
  - Eclipse: http://www.eclipse.org/cdt/ (this is used in a later Computer Science course as well)
  - Code::Blocks: http://www.codeblocks.org/
  - Xcode (Mac only): https://developer.apple.com/xcode/
  - NetBeans: https://netbeans.org/features/cpp/

    and from the Data Structures site, notes on using NetBeans with C and C++:

    http://chriseiffel.com/uncategorized/windows-c-compiler-how-to/

If you are having trouble deciding, ask the TAs and mentors what they use!

Remember: IDEs are useful for debugging with a graphical user interface. But, it is likely that no one will be an expert in the specific IDE you use. Practice and learn its features before starting to use it in class assignments.

48.3 Final Note

If you find additional resources that are useful, please feel free to let us know. We will include them here.