Python

Rev:2/21/2015

INTRODUCTIONS

- Instructor
- Students
  - What is your programming experience?
  - How long have you been programming?

PAPERWORK

- NLC registration
- RU roster
- Books (PDF)
  - Think Python, Allen B. Downey
  - Python for Informatics, Charles Severance

Resources

- Two books in PDF format:
  - Think Python
  - Python for Informatics
- DemoProgs – a collection of Python programs that concentrate on specific aspects of the language
- Samples - a set of screen shots designed to augment classroom discussion.
- LabsData – data and code to be used in lab exercises.
- Python Notes – a set of notes with explanations of various topics and pointers to articles containing more explanations. It is also a collection of methods used by various Python data structures.
- Python Notes Addendum – a short document containing pointers to intermediate-level Python topics.
- LabsDone – a collection of completed labs to be sent at the conclusion of the class.

The Zen of Python, by TimPeters:

- Beautiful is better than ugly.
- Explicit is better than implicit.
- Simple is better than complex.
- Complex is better than complicated.
- Flat is better than nested.
- Sparse is better than dense.
- Readability counts.
- Special cases aren't special enough to break the rules.
  - Although practicality beats purity.
- Errors should never pass silently.
  - Unless explicitly silenced.
- In the face of ambiguity, refuse the temptation to guess.
- There should be one-- and preferably only one --obvious way to do it.
  - Although that way may not be obvious at first unless you're Dutch.
- Now is better than never.
  - Although never is often better than **right** now.
- If the implementation is hard to explain, it's a bad idea.
- If the implementation is easy to explain, it may be a good idea.
- NameSpaces are one honking great idea -- let's do more of those!

Our Lab Environment

- Command line / Vim / Any other editor
- IDLE
  - Very modest IDE
  - Really primarily a calculator/editor
  - Interactive vs Editing windows
- Demo

PYTHON HELP

- Google!
- Interactive shell - help()
- At bash prompt: pydoc
  - e.g. pydoc -k string or pydoc module
- Windows Powershell
  - python –m pydoc --------
- Pydoc documentation
  - pydoc pydoc

PRINT STATEMENT

- The print statement is used to output one or more expressions separated by commas.
- General form:
  print expression1[, expression2,…,expressionN][,]
  where expression can be a variable, literal or any valid expression
- By default, each item is separated by a space.
- Automatically skips to next line when done, unless ended with ',‘
- Examples:
  a = 12.3
  print 'The value of', a, 'is in variable a.'
  printed result - The value of 12.3 is in variable a.
- print 'The value of', a, 'is in variable a.',
  print 'Is that enough?'
  printed result - The value of 12.3 is in variable a. Is that enough?

Basic Data Types

integer

float

string

boolean

None

Identifiers & Type

- Python uses weak data typing
- Data type is established with an assignment statement.
  - x = 10 (integer)
  - y = 12.34 (float)
  - z = ‘text data’ (string)
- You do not specify a size or precision for numbers.
  - Integers are essentially unlimited in size
  - Floats are all double precision
- Python is very generous in allocating memory to variables.
  - Demo program – sizecalc.py

Math Operators

**	Exponentiation
/	Division
*	Multiplication
+	Addition
-	Subtraction
%	Modulus (Remainder)
//	Floor Division

Compound operators (e.g., +=, -=, etc) work the same in Python as other languages.

Precedence is the same as all other languages

- Exponentiation
- Multiplication/Division (left to right)
- Addition/Subtraction (left to right)

Only parentheses change the order of execution.

Integer Math

- Math with integers produces an integer result in Python 2.
  - That is, 7/3 = 2 and 7//3 = 2.
- In Python 3, this changes.
  - Any division in Python 3 yields a float.
  - So, 7/3 = 2.3333 and only floor division gives 7//3 = 2.

Variable Assignment

- General assignments
  - x = 12.3
  - x = x + 1
  - y = x + 117
  - z = 12 * x / y
  - a = “This is a line of text”
  - b = “ and so is this”
  - c = a + b (string concatenation – do not use with numbers)
  - d = 20 * ‘-’ (string replication)

In Python, there can be more than one variable on the left side.

- x, y = y, x This swaps the values of the two variables
- Later, this technique will be used to unpack multi-valued variables into individual variables.
- See a0Assignments.jpg in Samples

Literals and Comments

- Comments work the same as most languages. A # is used as the delimiter.
- String literals can be enclosed in single quotes (') or double quotes (") if they are contained on one line.
- triple single-quotes (''') or triple double-quotes (""") are usually employed with literals or documentation that spans multiple lines.

```
"""
This is a sample program.  It shows how
documentation that spans multiple lines
can be delimited by triple double-quotes.
"""

""" The quotes can go on the same line also. """

print "Gold's Gym"       # No need for an escape
print 'He said, "Hello"' # No need for an escape
print 'He is 6\' 2" tall'  # The ' must be escaped
print """He is 6' 2" tall"""  # No need for an escape
```

Lab 01 - Formulas

- Create a program to solve these problems and print the results. Place each value specified by the problem into a variable before doing any calculations. Do not worry about the formatting of the answers. Your program should be such that you can change any of the numbers below and get a new answer.
  - You bought 125 shares of a stock at $25.32 and you sold it at $48.97. What is the profit?
  - A product with a price of $127.99 is going on sale. If the price is reduced by 16%, what will the new price be?

Formatting Data into Strings

General statement:

‘insert text here with {0} {1}'.format(variable, “literal string”)

Abbreviated general format of a formatting sequence:
{[seq#] ":" [width] [","] ["." prec] [type]} - See Python Notes for more detail

- The sequence number [seq#] is optional. If missing, variables/literals are formatted in the order given (python 2.7+).
- Width is used to expand a formatted item beyond the default. In the expanded width, numbers are right justified, text left justified.
- The ',' is used as a thousands separator in larger numbers.
- The ".prec" specifies the number of decimal places to display.
- The short list of valid types are s, d and f .
  - s – strings, d – integers, f – floating-point numbers

Formatting Data into Strings

General examples:

x = 'Some text {} more text {}'.format(12, 17.426)

Result stored as x – 'Some text 12 more text 17.426'

x = 'Some text {1} more text {0}'.format(12, 17.426)

Result stored as x – 'Some text 17.426 more text 12 '

x = 'Some text {0:5d} more text {1:7.2f}'.format(12, 17.426)

Result stored as x – 'Some text 12 more text 17.43' (Note rounding)

See examples in Sample folder - a2Formats.jpg and a3Formats.jpg

Lab 02

Go back and format the results from the last lab.

For more examples, see Python Notes and a2Formats.jpg and a3Formats.jpg in the Samples folder.

This is the newer method. The older formatting style in Python is a bit different, but quite similar.

Example: “There are %d %s in the box” % (5, “cookies”)

Further examples are in a1Formats.jpg in the Samples folder.

Variable Naming Conventions

- All names must begin with a letter or underscore. The remainder of the name can contain letters, numbers and underscores.
- These names cannot be a reserved word. (see Python Notes)
- Python standards are as follows:
  - Variables – lower_case_with_underscores
  - Constants – ALL_CAPS
  - Package and module names – lower case and short
  - Class names – CamelCaseWithoutUnderscores
  - Function names, method names, instance variables – same as variables
  - More detail available in PEP 8.

Continuations (from PEP 8)

- Python doesn't use a line delimiter, so statement continuations have to be managed.
- The preferred way of wrapping long lines is by using Python's implied line continuation inside parentheses, brackets and braces.
- Often, you will see the last enclosure on a line by itself.
- These should be used in preference to using a backslash for line continuation.
- Examples:
  - print 1234 + 5678 + 9012 + 3456 + \
    7890 + 1234 + 5678
  - print (1234 + 5678 + 9012 + 3456 +
    7890 + 1234 + 5678)
  - print (“I love to blah every morning. Then I blah around noon,”,
    “ and blah some more before bed.”)

Python Indentation

- The way Python uses indentation is unique
- Indentation shows where blocks of code begin and end
- Use 4 spaces per indentation level (not tabs).
- Mixing tabs and spaces will cause significant problems.
- IDLE will do this for you automatically
  - If it gets confused, it’s probably because of something you have done.
  - It replaces tabs with 4 spaces.

Comparison Operators

- == Equal
- != Not Equal
- > Greater Than
- < Less Than
- >= Greater Than or Equal
- <= Less Than or Equal
- not, and, or
- in, is

Making Decisions

- THERE IS NO CASE STATEMENT IN PYTHON!
- if or if/else or if/elif/else.
- Each option (if, elif, else) is followed by a colon (:) and then a suite of indented statements.
- For if/else or if/elif/else, one (and only one) suite will be executed.
- If a suite is defined, it must contain at least one statement.

Example:

```
if bank_account > 0:
        print 'Let's buy something'
else:
        print 'Go home instead'
```

- See c2If-Elif-Else.jpg in Samples.

Looping ? Times

- A way to loop: while/else statement
- Executes a set of statements in a loop until the while expression turns False, at which time the else block runs (if any)
- The else block runs even if the first test fails.
- Demo in samples (d1While.jpg)
  - Using a condition or True
  - Use of break

Getting Keyboard Input

- raw_input([prompt]) built-in function
  - The optional prompt string is the only argument
  - Data comes in as a string
  - The trailing newline (\n) is stripped.
- strings vs. floats and integers
- float() and int() functions
  - These functions strip leading/trailing whitespace (e.g., space, linefeed, tab).
- In Samples – bRaw_input-float-int.jpg

LAB 03

- Write a program asking the user for Fahrenheit temperatures to convert to centigrade in a loop. Convert each entry to a float before doing the calculation.
- Stop if raw_input() returns 'q' or 'Q'
- Remember:
  Fahrenheit = 9/5*Centigrade + 32
  Centigrade = 5/9*(Fahrenheit – 32)

- Save the results of this lab. We will be using it again.

Iteration, Iterable and Iterator

- Iteration is a general term for taking each item of something, one after another. Any time you use a loop, explicitly or implicitly, to go over a group of items, that is iteration.
- In Python, iterable and iterator have specific meanings:
  - An iterable is anything that can be looped over. As you will see, you can loop over a string, a file and a number of other objects.
  - An iterator is an object with state that remembers where it is during iteration, returns the next value in the iteration, updates the state to point at the next value and signals when it is done by raising a StopIteration condition.
- Whenever you use a for loop (covered next) and a number of other tools in Python, the next method is called automatically to get each item from the iterator, thus going through the process of iteration.

Looping N Times

- The for statement is used to loop a specific number of times.
- It has an else option that is rarely used.
- Executes a set of statements in a loop for the number of times you tell it, executing the same statements with different data
- Demo using range function.
  - range(start, stop, step) - you do not get "stop"
  - Use continue statement to end an iteration and start a new one.
- The for statement is used mainly as an iterator.
- The xrange function can be used as a generator

Loops Within Loops

- Loops can contain other loops.
- Example: (Yes, this is nonsense – it's just an example)

```
x = 10
while x > 0:
    for j in range(3):
        print x, j
    x = x - 1
```

- Break and continue statements work only in the loop in which they are executed.

LAB 04

- Create a times table
- It should be a matrix of the results of all possible single-digit multiplications

Ex:

1 2 3 4 5 6 7 8 9

2 4 6 8 10 12 14 16 18

And so on.

- You should use a for loop inside another for loop.
- Be sure to get the numbers properly aligned.

Importing Modules

- Much of Python’s capability is included in modules.
- You import only what you need.
- Use help to show modules. Interactive shell – help(‘modules’)
- Usually, you will want access to one or more functions within the module. (Ex: from math import sqrt)
- Sometimes, you might want access to variables in the module.
- Use pydoc on math, math.sqrt and random.randrange
- Demo with math and random

LAB 05 - Game

Create a game. Have the computer select a random integer between 1 and 100 inclusive. Then have the operator take successive guesses until they guess the correct number. At each try advise them whether the guess is too high or too low. If the guess is correct, tell them they won and tell them the number of attempts they took. Use the same format as the output sample to the right.

```
Enter your guess: 50
Your guess is too high
Enter your guess: 25
Your guess is too low
Enter your guess: 37
Your guess is too high
Enter your guess: 31
Your guess is too high
Enter your guess: 28
Your guess is correct!
You succeeded in 5 attempts.
```

EXCEPTIONS

- The try statement – handling exceptions.
- The statements are try, except, else and finally.
- try – includes the block of code in which specified exceptions will be trapped.
- except – specifies the exceptions themselves
- else – a block of code to be executed if there is no exception.
- finally – a block of code that will be executed whether or not there is an exception – good for cleanup operations.

CATCHING EXCEPTIONS

- See the example in the Sample folder (gTry_Except.jpg).
- A bare except statement will catch all exceptions.
  - It is not recommended
- Some common exceptions: NameError, ValueError, TypeError, IOError, StopIteration
- A complete list of exceptions can be found in the Python documentation

LAB 06

Re-implement the previous temperature-conversion lab to use a try statement to catch ValueError exceptions. Test by entering a non-numeric temperature.

Again, save the results of this lab. We will use it later

FILE I/O

A file is opened and assigned to a variable (object).

Examples: filein = open('file path and name', 'r')
fileout = open('file path and name', 'w')

Every other action is a method acting on the file object.

File records are usually read in a for loop which automatically ends at end of file. In this case, the readline method is not needed.

The valid methods for acting on a file object are read, readline, readlines, write, writelines and close.

More details on these methods are available in Python Notes.

Files as Iterables

- Files ARE iterables!
- Each record separated by a line feed (\n) can be read with a “for” statement.
- You do not need to test for end of file (null string).
  - See d3Read_File.jpg in Samples.
- Generic example:
  - fin = open(‘path to file/filename’, ‘r’)

    for linein in fin:
        process the record
    statements to execute upon file completion.
  - In this case, the variable linein will contain a new record for every cycle of the for loop.

LAB 07

Re-implement the previous temperature-conversion lab to replace the data source with a file instead of raw_input(). The file should be the data in the temps.dat file in your data folder. Be sure to account for any bad data that might be contained in this file. Instead of writing the results on the screen, place the results in a file that can be viewed through a text editor.

with Statement

- The with statement is used to run a suite of other statements under a “context manager” i.e. special methods __entry__() and __exit() are called to setup and takedown a “context”
- Common for doing I/O, which auto-closes the file handle
- Could be done with the finally option of a try statement, but with is more elegant and requires less code.
- Example using two files:

```
with open('input.file', 'r') as infile, open('output.file', 'w') as outfile
    for line in infile:
        do some processing
        outfile.write(some_string)
```

- At this point, there is no need to close the files as the with statement will take care of it – even in the event of an exception.
- See with.py in DemoProgs.

Iterables

- As we have seen, a file is an iterable and usable in a for loop.
- The next section on data structures deals with iterables as well.
- These are strings, lists, tuples, sets and dictionaries.
- Some of these are sequences, that is they are strictly ordered:
  - strings, lists, tuples
- The others are still iterables, but the order is undetermined.
  - sets, dictionaries

Strings – Basic Operations

- Strings are sequences. As such they support:
  - The len() function.
    - x = ‘himalayas’
    - len(x) is 9
  - The “in” operator (e.g., “red” in “Bred for speed” is True)
  - The “+” operator (e.g. “Hi” + “Ya” = “HiYa”)
  - The “*” operator in which the string is multiplied by an integer.
    - “Hi” * 3 = “HiHiHi” or 10 * “.” = “……”
- String comparison operators (See ASCII chart in Python Notes)
  - "abc" < "xyz“ (True)
  - "ABC" == "abc“ (False)
  - “abc” < “ABC“ (?)

Strings – Slicing

- In Python, slices are used instead of a method (i.e., substring)
- String Slicing - string[start:stop:step]
  - start – the first or only item we want (starting from zero)
  - stop – the first item we don't want
  - step – an increment other than 1 which is the default
  - If x = 'himalayas'
    - x[0] is 'h', x[3] is 'a', x[6] is 'y', (you get only one character)
    - x[1:3] is 'im', x[:3] is 'him, where start defaults to zero.
    - x[6:] is 'yas', where stop defaults to include the last character
- Slices can be negative. There is no -0.
  - x[-1] is 's', x[-3] is 'y', x[-9] is 'h'
  - x[:-1] is 'himalaya', x[-6:-1] is 'alaya'
- Steps – x[2:8:2] is 'mly' You do not get the end point!

LAB 08

Review the file tmpprecip2012.dat. It is laid out as follows:

Columns	Content
1 – 2	Month
3 – 4	Day
5 – 8	Year
9 – 13	Precipitation in the format dd.dd (inches)
14 – 16	High Temperature

The data is in chronological order starting with 1/1/2012 and working up to 12/31/2012. Accumulate the number of days with measurable rainfall and the precipitation total for the year and print them out. Be sure to account for invalid data.

Iteration

- Strings are iterables.

```
x = 'red'
for i in x:
    print i
```

Result:

```
r
e
d
```

- See h1StringSlice.jpg in Samples

Strings - Methods

- Methods are just functions on an object
- All string methods produce a result. Nothing is done in place.
- Strings cannot be changed in place. (Immutable)
- Every time you make a change to a string, a new string is created in a new location.
- See Python Notes for a complete list of string methods.
- Also, see h2StringMethods.jpg in Samples.

Strings - Methods

- Invoked with dot notation
  - x = ‘himalayas'
  - x.upper() returns ‘HIMALAYAS’
  - x.find("ya") returns 6
  - x.find(“az") returns -1
  - x.count("a") returns 3
  - x.startswith("hi") returns True
  - x.endswith("az") returns False
  - x.isalpha() returns True
  - x.isdigit() returns False

LAB 09

In your data folder, you will find a file containing the text for the book, “Alice in Wonderland. Read the entire file into memory (read method instead of readline). Scan this text counting all of the letters. Keep a separate count for all occurrences of the letter ‘e’. Print a line showing the percentage of all letters that are e's

Strings

- Built-in functions chr() and ord().
- ord() returns the decimal equivalent of an ASCII character.
- chr() returns the ASCII character corresponding to the integer provided.
  - x = ‘himalayas’
  - mynum = ord(x[3]) returns the integer 97 as mynum
  - letter = chr(mynum) returns the character ‘a’

Reading a Web Page

- Use the modules urllib or urllib2.
- variable1 = urllib.urlopen('url address')
- variable2 = variable1.read() # reads the whole page

  or
- for line in variable1: # reads one line of the web page at a time
- Basically used for parsing html/screen-scraping.

- The above are fine for very simple operations.
- For more demanding requirements, consider using Requests

Lists

- Python’s implementation of tables/arrays.
- Can be multiple dimensions.
  - We will only deal with two at most.
- Lists are produced using square brackets.
- Lists can contain items that are all the same type or many different types.
- Like strings, lists are accessed with integer indexes
- Unlike strings, lists can be changed in place (mutable).

List Operations

- x = [12, 3, 124, 56, 2]
- len(x) is 5
- x[1] is 3, x[-1] is 2
- The LIST function makes a list out of any iterable.
- The RANGE function creates a list
- y = range(1, 10 ,2)
  - y is now a list = [1, 3, 5, 7, 9]
- See iList1.jpg in Samples

List Operations

- x = [] creates an empty list
- y = 5 * [0] creates a list of five zero integers
- z = [2, 3, 4] + [5, 6] is [2, 3, 4, 5, 6]
- Lists are iterables!

```
x = [1, 34, 12]
for i in x:
    print i
```

Result:

```
1
34
12
```

- x = [1, 34, 12]
- x[0] += 1
- x is now [2, 34, 12]

Lab 10

Plan and execute the following program. Create a function to simulate the rolling of a pair of dice. Call this function 10,000 times accumulating the results of each roll in a list. When finished. Print the percentage of times each possible roll occurred along with the total number of rolls. Compare your results to the following mathematically derived results:

2	2.78%
3	5.56%
4	8.33%
5	11.11%
6	13.89%
7	16.67%
8	13.89%
9	11.11%
10	8.33%
11	5.56%
12	2.78%

List Methods

- In place change vs return – be careful
- In place change (fruitless/void)
  - append() vs. insert() vs extend()
  - sort() vs. reverse()
  - remove()
- Produce a return (fruitful)
  - count(), index()
- Both – in-place change and a return
  - pop()
- See iList2.jpg in Samples.

Changing a List

- my_list = [1, 2, 3]
- my_list.append(4) # ok
- my_list.extend([4]) # ok
- my_list.insert(0, 4) # ok
- my_list = my_list + 4 # error
- my_list = my_list + [4] # changes locations – not a good idea.
- my_list = my_list.append(4) # wipes out list
- my_list.append([4]) # inserts a list within the list
- See list_times.py in DemoProgs

List Operations

- x = [12, 3, 124, 56, 2]
- Built-in functions such as SUM, MAX and MIN can operate on a list of numbers.
  - e.g., max(x) is 124, sum(x) is 197
  - MIN and MAX can operate on non-numerics as well.
- Unpacking a list:
  - a, b, c, d, e = x # unpacks the list
  - See list_unpk.py in DemoProgs

LAB 11

Read the trees.dat file putting each valid element into a list. The file contains the height in even feet of a large sample of California coastal redwood trees. When finished, use only built-in functions and normal math equations to produce a report on the screen showing:

- the number of trees,
- the average height of the trees to one decimal place,
- the height of the tallest tree, and
- the height of the shortest tree.

Lists – Removing Elements

- pop() if index known or last element
- remove(element) if index not known
- del operator
- del with slice notation
- del is used for many things, not just lists

Two-Dimensional Lists

- Lists can be many dimensions. We will deal with two.
- List 1 List 2

eggs
milk
bread

eggs	2	dozen	free range
milk	3	quart	2 percent
bread	1	loaf	whole wheat

- x =[[‘eggs’, 2, ’dozen’, ‘free range’], [‘milk’, 3, ‘quart’, ‘2 percent’], [‘bread’, 1, ‘loaf’, ‘whole wheat’]]
- x[1][2] = ‘quart’, x[0][1] = 2, x[2][0] = ‘bread’
- How do they sort? Demo basics with iLists5.jpg in Samples
- Initialize a two-dimensional list – see Samples (iLists4.jpg, iLists4alt.jpg)

If I want 1 more dozen eggs, how do I add 1 to the eggs?

Sorting

- The sort and reverse methods operate only on lists
- The sorted and reversed functions operate on any iterable.
  - Also, they produce a result and leave the original object intact.
  - Sorted produces a list while reversed produces an iterator which can be used in a for loop or transformed into a list with the list function
- See the Python Notes for a link into more detailed sort documentation.
- See DemoProgs (sortitem.py and sortitem2.py).

Lab 12

Read the data in tmpprecip2012.dat and create a two-dimensional list containing all the data that will allow you to print a report by month of the following:
Average high temperature,
Maximum high temperature,
Minimum high temperature

Once that works, try your program on tmpprecip.dat. It contains over 100 years of daily data.

The format of the data is repeated here:

Columns	Content
1 – 2	Month
3 – 4	Day
5 – 8	Year
9 – 13	Precipitation - format dd.dd (inches)
14 – 16	High Temperature (integer)

2012 report →

1	68.4	78	53
2	66.2	85	43
3	76.2	88	49
4	85.2	95	76
5	87.9	96	71
6	95.3	106	88
7	94.9	100	84
8	98.5	103	91
9	90.5	99	73
10	80.5	90	59
11	74.3	86	53
12	68.8	82	48

Deep vs Shallow Copy

- Equivalent or identical ?
- Use id() function or *is* operator to find out.
- When identifiers point to the same value (object) in memory.
- What do x and y equal in each case? Are they the same thing?
  - x = 12; y = 12; x += 1 immutable example
  - x = [1, 2]; y = x; y.append(3) mutable example
- Use deepcopy to avoid aliases. Used mainly with lists and dictionaries but can be used with functions and classes also.
- See deepcopy.py in Samples

Tuples

- Tuples are basically immutable lists and are iterable.
- Defined directly by parentheses or tuple function.

```
x = (1, 2, 3)
y = tuple(iterable)
z = (12,)  a single item tuple
```

  - Iterable can be any type, even another tuple

- x = () is an empty tuple.
- Tuples are typically used for value passing to avoid clobbering
- Tuples show up in lots of places.

Tuple Assignment

- a, b = (1, 3) "unpacks" a tuple
  - Left side: variables (identifiers) to receive values
  - Right side: tuple or sequence
  - Number of identifiers must equal number of tuple objects
  - a, b, c = "123" ?
- x = (1, 2, 3)
- y = (4, 5, 6)
- z = x + y gives (1, 2, 3, 4, 5, 6)
- There are no modification methods for tuples.

How Are Tuples Used?

- Try x = divmod(7, 3)
  - What data type is x? What values does it contain? Why?
  - Try x, y = divmod(7, 3) What are the data types now?
- Zip two lists together.
  - x = [1, 2, 3]; y = [4, 5, 6]; z = zip(x, y) What is z now?
  - Get used to this structure. We will see it in dictionaries.
  - Can I sort this result? How?
- A number of other ways which we will see.

Functions

- Python function definitions can be defined independently and called directly. (Sample - f1Functions.jpg)
- The def/return statements delimit a function.
- The statement suite of a function is indented just like all other blocks of Python code.
- The return statement is not required. If not present, the function returns a None value.
  - The end of the function is detected through indentation changes.
- The return can identify one object to be returned.
- Functions are 'called' directly with arguments
- Functions enable reusability even without creating classes.
- Variables defined in a function are local to that function. Variables defined in the main program are known globally.

LAB 13

Re-implement a previous temperature-conversion lab to use a function to do just the conversion formula.

Function name should be *f_to_c()*

Save this lab for use in subsequent labs including the Lambda lab.

IDENTIFIER SCOPE

- Scope is how widely the identifier can be seen in other parts of the code
- Local scope
- Global scope
- 'Built-in' scope
- Example in Sample folder – f2Functions global-local

Variable Parameter Collectors

- Tuples as function arguments
- def fn(*varname) varname is traditionally “*args”
- *varname parameter receives all excess positional arguments
- If you have a variable number of parameters, how do you test them for validity?
  - Use isinstance built-in function to test argument type.
  - example: if isinstance(x, type) where type is int, float, str, etc.

Variable Parameter Collectors (cont.)

- Functions have a declaration (also called a “prototype”) which gives them a name and some “arguments” to work on:

  *def name(positional args, keyword args, *args, **kwargs)*

- Keyword arguments assign a default value (e.g., x=14).
- If the function is called without specifying a particular keyword, the default value stays in effect.
- If the keyword is specified, the default is overridden.
- If keyword arguments are passed that have no default defined, they are accumulated in kwargs as a dictionary.

LAB 14

Change the program you just created (with a function to do temperature conversions) to accept a variable number of temperatures per function call and process all of them. Print the collector argument and its type. Use the isinstance function to verify the type of each parameter as you iterate through them. Each parameter should be either int or float. Reject all others. Test with invalid data.

Example:

a) fahrenheit_to_centigrade(72, -10.5, '2e', 111, 55) # function call

Have the function parse/test the arguments and print all results.

Save this program for use in the command-line lab

Command-Line Arguments

- Python allows you to access the command-line arguments passed to the script.
- The argv variable (a list) in the sys module contains the command-line arguments passed to the script.
- The first argument is always the name of the script.
- The actual arguments passed start with the second argument.
- Example:

```
from sys import argv
for x in argv:  # Remember, argv is a list
        process each argument
```

LAB 15 – Command Line

Change the previous lab to accept a variable number of parameters (temperatures to convert) from the command line. Parse/test all the arguments and print all results. Repeat the actions you performed before using different tools as necessary.

Example:

python parms.py 72 -10.5 111 55 #command line parameters

Dictionaries

- The purpose of a dictionary is to associate a key with a value for very fast lookup.
  - Dictionaries are much more efficient in some circumstances than lists.
- Dictionaries can be created in two ways:
  - using the dict built-in function
  - using braces – x = {} is an empty dictionary.
- Keys can be any immutable type : numbers, strings. tuples.
- Keys are hashed for fast lookups.

Dictionaries

- Examples:
  dict_01 = dict() creates an empty dictionary
  dict_02 = {sun: 1, mon: 2, tue: 3, and so on}
- Order of items unknown
- KeyError access exception
- len() tells you the number of key:value pairs
- By default, *in* operator works on keys only
- keys(), values(), and items() methods unload all or parts of the dictionary.
- Review the sample - jDictionary1.jpg

Dictionaries as Counters

- Key = item to count
- Value = count
- What is a *histogram*?
- Using dictionaries to implement a histogram
- In “Python for Informatics” read about dictionaries as counters (page 109)

LAB 16

The text for, “Alice in Wonderland” is located in your data file. Use it as input to this exercise. Parse the file and identify each unique character and the number of times it is used. Do not differentiate between differently-cases letters. Exclude all whitespace characters from this exercise. A string of whitespace characters can be imported from the string module. Accumulate this information in a dictionary.

Create a report displaying each character and the number of times it is used. The list should be in order from most to lease occurrences.

You might want to review the sorting techniques in the DemoProgs folder.

Strings (Again)

- split() - delimiters.
- What do these operations do?

  linein2 = 'first:second:third:fourth:last‘

  line2 = linein2.split(‘:’)

  linein = “\nA serious error has occurred on your watch\r\n"

  line = linein.split()
- String module has useful variables. (e.g., punctuation)
  - In Python shell enter help('string'). The variables are defined at the end.

LAB 16 - Original

The text for, “Alice in Wonderland” is located in your data file. Use it as input to this exercise. Parse the file and identify each unique word and the number of times it is used.

Print a histogram of the top 20 words in the word frequency dictionary. Print the word (key), then from 1 to 20 asterisks proportional to the value (frequency).

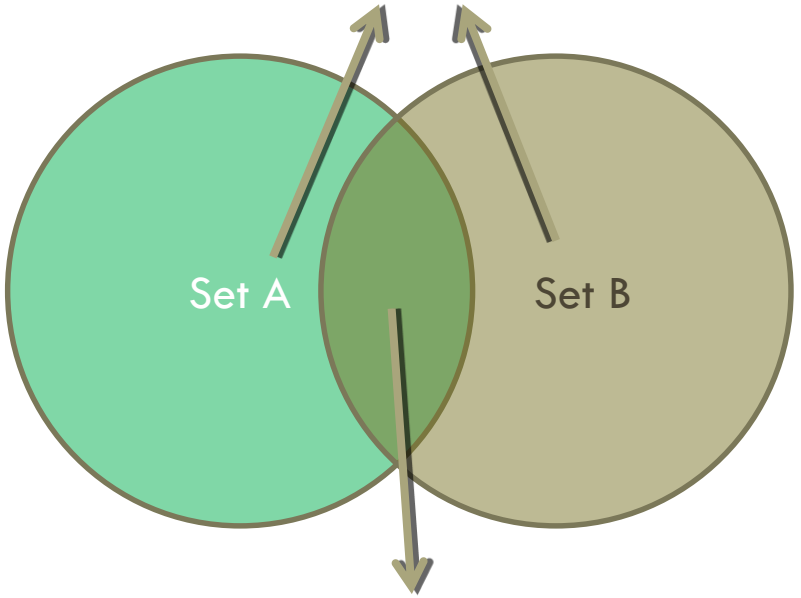
You might want to review the sorting techniques in the DemoProgs folder.

Sets

- Sets are an additional type available in Python.
- Sets are unordered and contain only unique items.
- As with most iterables, the len built-in function is operable.
- Demo
  - set1 = set('himalayas') will contain h, i, m, a, l, y, s
  - set2 = set([1, 2, 3, 2, 6, 3, 1, 5]) will contain 1, 2, 3, 6, 5
  - set = {12, 2, 104, 18} creates a set in Python 2.7 (not 2.6)
- union: sum of both sets with duplicates removed
- intersection: in both sets
- See the set methods in Python Notes

Sets

Union Set A and Set B (No duplicates).
All unique items in both sets



Intersection Set A and Set B (No duplicates).
all items that are in both sets.

See kSets.jpg in Samples

LAB 17

First, create a list of 50 unique numbers. Use the sample function from the random module to create the following sets:

- a control set of six numbers
- a list containing 100 sets each having six numbers.
- All of the numbers in the above sets should be selected from the list of 50 numbers using the sample function.

Compare all sets in the list to the control set and print the number of times at least 2 matches occurred.

Review help('random.sample') for clarification. Be sure to ignore the 'self' parameter as you did with randrange.

List Comprehensions

- Convenient way of initializing a list with an arbitrary expression
- General Format:
  [expression for expr1 in iterable1 [if condition1]
  for expr2 in iterable2 [if condition1] …]
- All values of expression that meet the optional conditions are included in the final list.
- Nesting is not recommended as it makes the code difficult to read.
- Examples:
  [x*2.5 for x in range(5)]
  result - [0.0, 2.5, 5.0, 7.5, 10.0]
- [x*2.5 for x in range(5) if int(x*2.5) == x*2.5]
  result - [0.0, 5.0, 10.0]

Dictionary and Set Comprehensions

- Shorthand way of initializing these data structures.
- Allows control of number of elements, and built-in filtering all in one operation.
- The alternative is to use for loops which are sometimes more readable.
- Examples (can you guess which is which?
  - y = {x for x in xrange(2)}
  - z = {x: chr(x+65) for x in xrange(20)}

LAB 18

Use the range function to create a list containing the numbers:

[2, 5, 8, 11, 14, 17]

- Use list comprehensions to:
  - Create and print a new list with only the even numbers from the original list.
  - Create and print another new list containing the square of the original numbers.

Lambda Functions

- Lambda functions are anonymous, one-line functions.
- They are used to avoid littering your program with small functions.
- A lambda function can take multiple arguments and produce one result.
- Examples:
  a1 = lambda x : x ** 2
  a1(5) produces 25
  f = lambda x, y : x + y
  f(14, 12) produces 26

- Run these examples in the Python shell.

LAB 19 - Lambda Functions

Take the initial temperature conversion lab containing a function and replace the function that does the conversion with a lambda function to do the same thing.

Filter, Map, Reduce

- All of these can be done using techniques we have learned.
- They can also be done using built-in functions and lambdas.
  - Filter(), Map(), Reduce() and Lambda functions
- The initial developer of Python, Guido van Rossum, wanted all three of these built-ins as well as lambda functions removed from Python 3.0. He was unsuccessful.
  - Reduce() did get moved to the functools module and has to be imported.
- Example: (Try in the shell)
  - y = [2, 6, 9];
  - x = map(lambda z : z **2, y)

LAB 20 – Filter, Map Reduce

Use the range function to create a list containing the numbers:

[2, 5, 8, 11, 14, 17]

- Use map(), filter() and reduce() in combination with lambda functions to accomplish the following:
  - Create and print a new list with only the even numbers from the original list. (filter)
  - Create and print another new list containing the square of the original numbers. (map)
  - Create a result showing the sum of all the original numbers. (reduce)

yield Statement

- yield is similar to return, but suspends execution of the called function instead of ending the function
- On the next call to the function, yield picks up where it left off, with all identifier values still holding the same values (a return loses its identifier values)
- Use pydoc or help for more information on yield

Generators

- Generators are functions that return a “generated” (according to your algorithm) set of values (see PEP255)
- A generator is identified by a function that uses the yield statement
- Generators are just lazy iterators that don’t pre-generate the results until needed.
- Called with next() just like any generator
- Saves memory: does not pre-build returned object
  - Ex. range vs xrange
- See generator_test.py in DemoProgs

Lab 21

- Write a generator that returns an increasing integer
  if the integer is even or the string "odd" if the integer is odd.

- Test your generator by calling next() 8 times. The generator itself
  should never raise StopIteration.

Generating Decimal Numbers

- Generators can be used in place of the range function to generate floats. Working with floats requires caution.
- Implement the following program. Does it work as expected? If not, why?

```
def frange(start, stop, step):
    i = start
    while i < stop:
        yield i
        i += step

for x in frange(0.5, 1.0, 0.1):
    print x
```

Decimal Module

- The previous slide shows that many floating-point numbers are not precisely represented.
- For this reason, most languages advise you not to use floats for tracking monetary values.
- Within Python's decimal module, the Decimal class allows you to avoid this problem.
- The demo program shows how this works at its simplest. (generatordecimals.py)

Lab 22 - sys and os Modules

- These are extremely large modules containing many diverse functions.
- We will look at just a few of the capabilities.
- Use pydoc to list the documentation for the following, and then create some data on which to test them:
  - os.getcwd
  - os.path.dirname and os.path.basename
  - os.path.exists
  - os.path.isdir and os.path.isfile
  - sys.exit and sys.argv
- os.environ is a dictionary. Import it and try to print it neatly so you clearly see what it contains.

Links to formal Python documentation for these modules:

http://docs.python.org/library/sys.html#module-sys

http://docs.python.org/library/os.html#module-os

Namespaces and Scoping

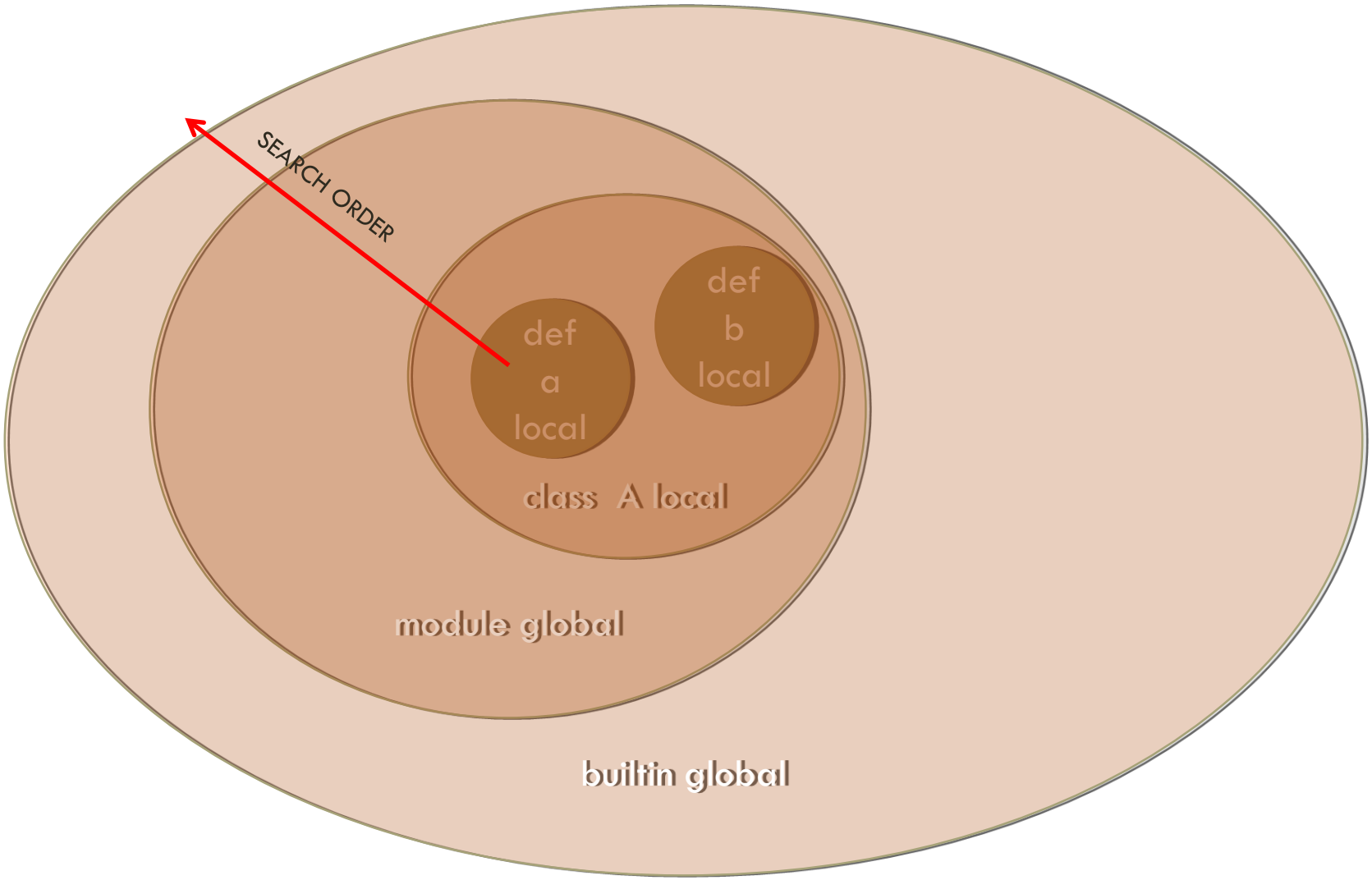
- Everything in a Python program is an “object”.
- Objects have state and behavior.
  - Python stores an object's state in a variable. (x = 'abcde' : a string)
  - It exposes an objects behavior through methods. (x.isalpha() = True)
- **Block**: any Python code executed as a unit e.g. a module, function, class (to be covered next) and scripts.
- Every block of code has its own namespace where all the variables (objects) for that block are defined and tracked.
  - The structure used for this information is a dictionary.
- It is possible to use the same variable name in different blocks.
  - But is not a good idea.

Namespaces and Scoping

- **Scope**: the space where a identifier is visible. Contained blocks can see identifiers of outer blocks
- You start your script and it begins to execute.
  - The portion of your script/program where execution begins is the main portion of your program. As you have seen, the internal name for this block of code is "__main__".
  - Everything defined in __main__ is global. That is, all of variables defined in __main__ are known (visible) everywhere; in all functions, classes and modules.
  - That does NOT mean you can effectively modify that data from another block (e.g., a function)
- The objects **imported from** any module are known globally also.
  - You need to be aware of potential conflicts.
- The objects in functions and classes are not known globally

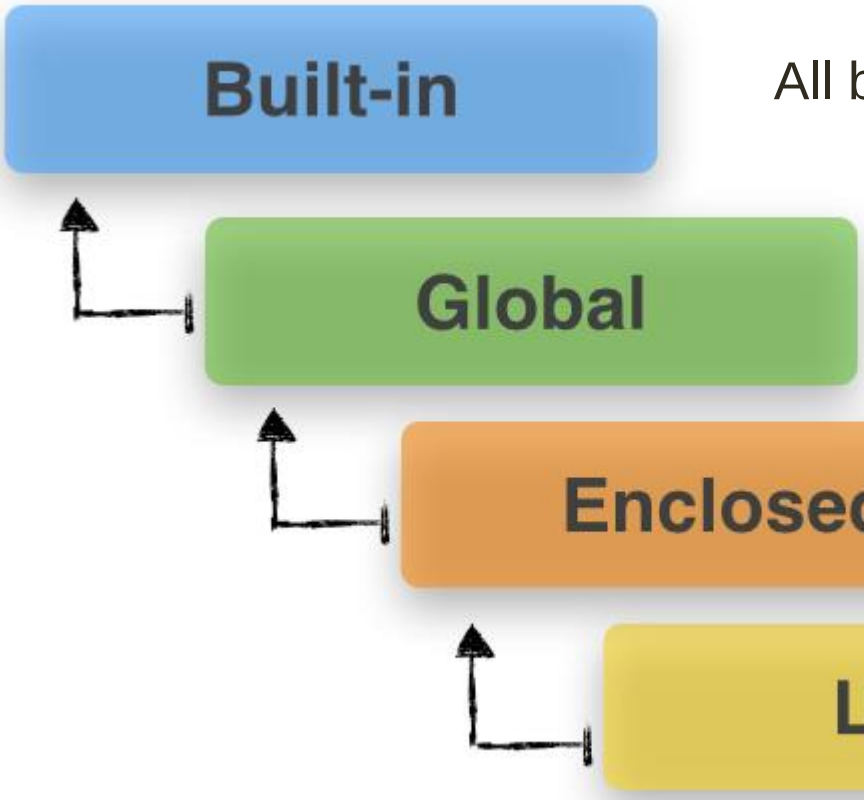
Namespace Diagram

There is a local namespace for each code block.



Namespace Hierarchy

Another way of looking at it



All built-in functions

Imported modules and all objects in __main__

Higher Level Functions/Methods/Classes

Lowest Level Functions/Methods

In DemoProgs, run lcl_glbl.py, lcl_glbl2.py, lcl_glbl2pp.py and lcl_glbl3_legb.py

Namespaces and Scoping Summary

- Variables are names (identifiers) that map to objects. A *namespace* is a dictionary of variable names (keys) and their corresponding objects (values).
- A Python statement can access variables in a *local namespace* and in the *global namespace*. If a local and a global variable have the same name, the local variable is shown over the global variable.
- Each function has its own local namespace. Class methods follow the same scoping rule as ordinary functions.
- Python assumes that any immutable variable assigned a value in a function is local. Therefore, in order to assign a value to a global immutable variable within a function, you must first use the global statement. (not recommended)
- The statement *global VarName* tells Python that VarName is a global variable. Python stops searching the local namespace for the variable.

Object-Oriented (OO) Programming

- **Class:** A user-defined prototype for an object that defines any instance of the class. The attributes are data members (class variables and instance variables) and methods, accessed via dot notation. (e.g., a cookie cutter)
- **Instantiation :** The creation of an instance of a class. (e.g., creating a cookie)
- **Instance:** An individual object of a certain class. (e.g., the actual cookie)
- **Instance variable:** A variable that is defined inside a method and belongs only to the current instance of a class.
- **Method :** A special kind of function that is defined in a class definition.

Class Example (code in LabsData)

```
class BankAccount(object):  # Top tier class
    def __init__(self):   # This method runs during instantiation
        self.balance = 0  # instance variable
    def withdraw(self, amount):  # a method
        self.balance -= amount
        return self.balance
    def deposit(self, amount):     # another method
        self.balance += amount
        return self.balance


a = BankAccount()  # Create an instance of Bankaccount
b = BankAccount()  # Create another instance
a.deposit(100)    # Deposit $100 into account represented by
                  # variable a.
```

The Reason You Need to Use Self

- Python decided to do methods in a way that makes the instance to which the method belongs be passed automatically, but not received automatically:
- The first parameter of methods is the instance the method is called on. (e.g., a.deposit(100) will pass the object (a) and 100 to the deposit method in the example above.)
- That makes methods essentially the same as functions.
- It leaves the actual name to use up to you (self is the convention, and people will generally frown at you if you use something else.).
- Self is not special to the code, it's just another object.

Classic vs. New Classes

- The “classic-style” class goes away entirely in Python 3.X
- Always use “new-style” classes in new code
- Beware: Python 2.X class statement *still defaults to classic classes* unless a new-style class or object is explicitly used as the base class.

Class Example

- How do I access the balance of an account?

```
a = BankAccount()
b = BankAccount()
a.deposit(100)
print a.balance   # Balance is an instance variable
```

- In this case we are printing the balance associated with the account pointed to by the variable a.
- How do I make comparisons?

  if a == b: Does not work. We want to compare balances

  if a.balance == b.balance: Will do the desired comparison.
- Often we want to do more complex comparisons.
- These are best done in a method.

Comparing Instances

- We can create our own method.

```
def compare(self, other):
        if self.balance == other.balance:
            return True
        else:
            return False
```

- if a.compare(b): self will receive a and other will receive b
  - This will come back either true or false
  - But we could use one of the "magic methods" instead.

Python Magic Methods

- Python has a large number of methods that operate "behind the scenes" so to speak.
- The names of all of these methods include double underscores
- Several commonly-used of these methods are:

  __init__ called when an instance is created.

  __del__ called when an instance is deleted.

  __str__ called when an instance is printed (for example)

  __cmp__ called when instances are compared (not in ver 3)

  __eq__, __ne__, __lt__, __gt__, __le__, __ge__
- The last six methods take the place of __cmp__ and are much more flexible. Use these instead of __cmp__.
- Python Notes Addendum has a pointer to a complete list of these methods.

Class Example (__str__)

```
class BankAccount(object):  # Top tier class (super class)
    def __init__(self):   # This method runs during instantiation
        self.balance = 0  # instance variable
    def withdraw(self, amount):  # a method
        self.balance -= amount
    def deposit(self, amount):     # another method
        self.balance += amount


a = BankAccount()  # Create an instance of Bankaccount
b = BankAccount()  # Create another instance
a.deposit(100)    # Deposit $100 into account represented by a.
print a                # This will not work as desired.  The default
                       # __str__ method is insufficient.
```

Lab 23 - Class Example

- Try the code on the previous slide. What happens?
- Add this code to your program:

```
 def __str__(self):
        print '__str__ method entered'
        return 'The balance for this account is ${:,.2f}'.format(
            self.balance)


a = BankAccount()  # Create an instance of Bankaccount
b = BankAccount()  # Create another instance
a.deposit(2100)
print a
```

Class Example

- Create a copy of your program we just modified.
- Try this in the main program:

```
a = BankAccount()
b = BankAccount()
a.deposit(100)
b.deposit(100)
if a == b:
    print 'Accounts are equal'
```

- Why doesn't this work?

Lab 24 - Class Example

- When you compare, __cmp__ is called automatically and it will blindly compare two instances which will never be equal.
- To override __cmp__, implement one or more of the newer magic methods – in our case __eq__.
- Add this code to your program:

```
 def __eq__(self, other):
        print '__eq__ method entered'
        if self.balance == other.balance:
            return True
        return False


a = BankAccount()
b = BankAccount()
a.deposit(2100)
b.deposit(2100)
if a == b:
    print 'The two accounts have the same balance'
else:
    print 'The balances are different'
```

Class Variables

A variable that is shared by all instances of a class. Class variables are defined within a class but outside any of the class's methods. Class variables aren't used as frequently as instance variables.

Class Variables

- Change your program to add this code:

```
class BankAccount(object):
    acct_cntr = 0        # class variable
    def __init__(self):
        self.balance = 0    # instance variable
        BankAccount.acct_cntr += 1   # Accessing a class variable
:
:
a = BankAccount()
print 'Number of accounts -', BankAccount.acct_cntr
b = BankAccount()
print 'Number of accounts -', BankAccount.acct_cntr
```

- Note how class variables are accessed.

Lab 25

- Add the following statements to the end of your program

```
del a
print 'Number of accounts -', BankAccount.acct_cntr
```

- What is the problem and what has to change?
- Implement the magic method needed to get the correct result. if necessary, go back a few slides to the list of these methods.

Inheritance

- Inheritance describes the transfer of the characteristics of a class to other classes that are derived from it.
- Example:

```
class A(object):
    def mthd1(self):
        do something
    def mthd2(self):
        do something
class B(A):
```

- Class B has inherited all the methods and instance variables of class A.

Overrides

- Example:

```
class A(object):
    def mthd1(self):
        do something
    def mthd2(self):
        do something
class B(A):
    def mthd2(self)
        do something different
```

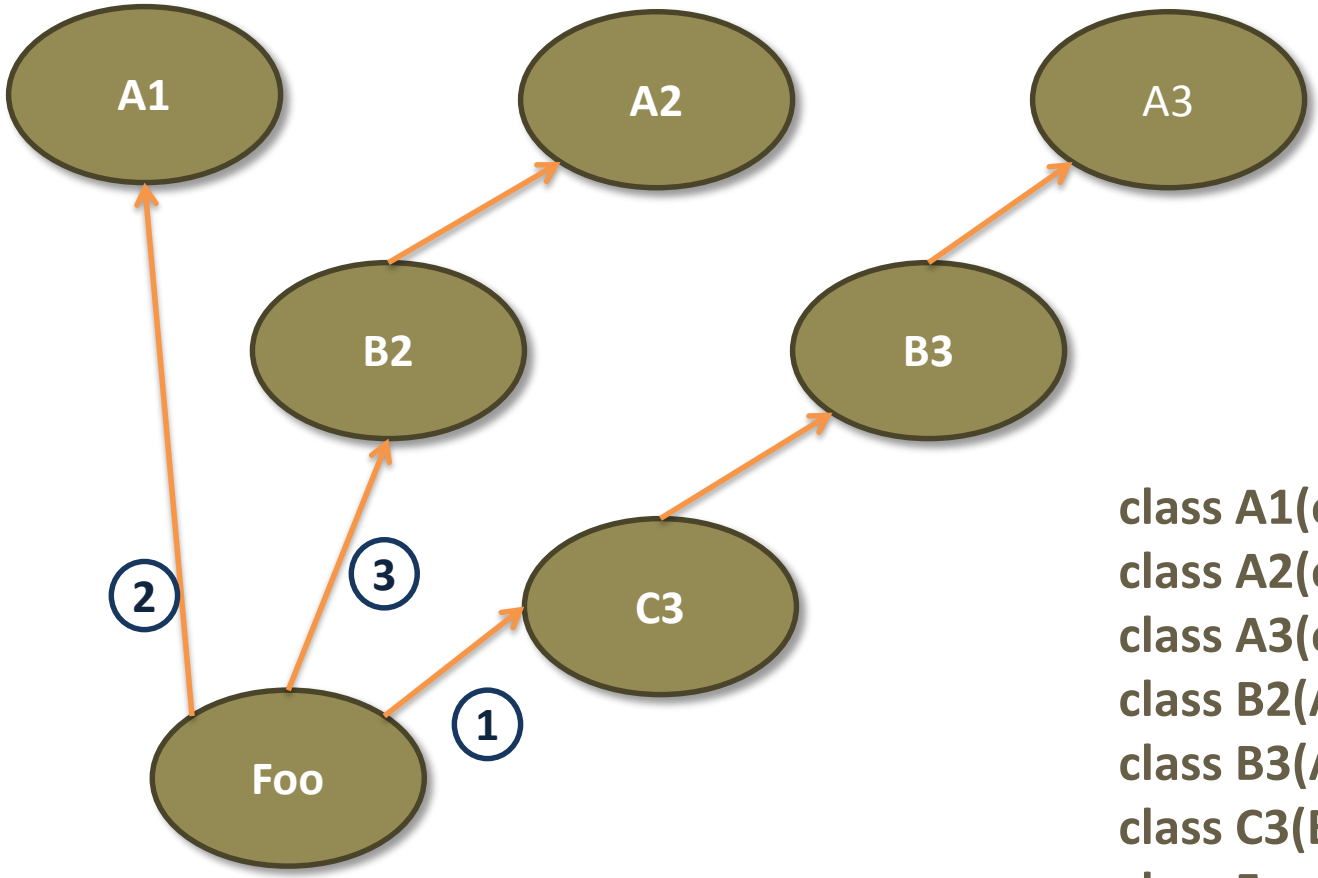
- Class B has inherited all the methods and instance variables of class A. In addition, Class B replaces mthd2 with its own version of that method.

Method Resolution Order (MRO)

- For simple class structures, MRO can be summarized as left to right and depth first but common ancestor classes are only checked after all of their children have been checked. Given these new style classes:

```
class A1(object): pass
class A2(object): pass
class A3(object): pass
  class B2(A2): pass
  class B3(A3): pass
  class C3(B3): pass
class Foo(C3, A1, B2): pass
```

MRO



class A1(object): pass
class A2(object): pass
class A3(object): pass
class B2(A2): pass
class B3(A3): pass
class C3(B3): pass
class Foo(C3, A1, B2): pass

Lab 26

- Change the previous program to allow, in addition to the normal account, an account with a minimum balance. Do this through a new class that inherits from the original. Make sure the new class does the following:
  - Provides for an initial deposit as well as a specified minimum balance on account creation.
  - Does not permit a withdrawal that takes the balance below the minimum. Prints an error message instead.
  - When an instance is printed, print the minimum as well as the balance for minimum-balance accounts.
  - Confirm that the deposit method works for the new class.
- Note – class variables are referenced through the original class

raise Statement

- The raise statement is used when code wants some attention because of an event. The attention-getting device is an Exception object
- Although there are more complex forms of raise, the example below is the simplest:
  raise [Exception [, arg]] – where arg is usually a string
- If a problem is discovered during an instantiation (__init__), you can cancel the instantiation by raising an exception.
- See Python Notes Addendum for a link to a more comprehensive explanation.
- See raise1.py and raise2.py in DemoProgs

Lab 27

Change the previous program to prevent a minimum balance account from being created without an initial deposit that meets the required minimum. Capture the error in the main program and print an appropriate message indicating the problem.

Debugging Using PDB

- Some Vocabulary:
  - Single step: run one line of code and stop
  - Step in: trace into a function
  - Step out: return from tracing a function
  - Breakpoint: run normally until here, then stop
  - Watch list: monitored variables

- Set breakpoints in the code
  - import pdb to get the software loaded
  - pdb.set_trace() – insert this code at every desired breakpoint
  - This does not preclude using the commands on the next slide.
- Run program under pdb.
- From command line enter pdb modulename and use the commands on the next slide. (Windows – python –m pdb module)

Common PDB Commands

- h(elp) list commands or display doc for a specific command
- b(reak) set a breakpoint or (no arguments) list all breakpoints
- s(tep) step into a function. Using n will bypass the function
- n(ext) exceute the next statement.
- r(eturn) step out of a function
- c(ontinue) execute normally stopping at any breakpoints
- l(ist) list the source statements adjacent to current location
- p expression print this value to the screen. Usually a variable
- q(uit) get out of pdb altogether
- c(lear) clear a specific breakpoint. No argument clears all breaks
- !expr change the value of any variable while at a breakpoint. The ! is needed to avoid confusion with a command.

Independent Lab

- From command line, execute python pdb1.py
  - Note what happened when pdb is imported and set_trace is used.
  - At the break, issue h to get a list of commands.
  - Also, print the value in variable a and then variable b.
  - Why is there a problem with variable b?
  - Change the value of variable a to 'zzz'
  - Enter command n and step through the rest of the program.
- From command line, execute python pdb2.py
  - Set breakpoints at lines 8 and 11.
  - Execute (command c) the program until the break. Print variable a.
  - Step through the program using command n.
  - Be sure to use the s command to step into the function.
  - Continue to the breakpoint with a c command. Issue a b command, then a clear (respond y) then b again. Note all breakpoints cleared.
  - Print variables s1, s2 and s3 and issue a c command to finish.

Future

- __future__ allows you to use certain elements of version 3 while still in version 2
- Examples:

7/3 = 2

from__future__ import division

7/3 = 2.33333

print 'Why wait for version 3?',

from __future__ import print_function

print('Why wait for version3?', end = ' ')

- See future1.py and future2.py in Demo Progs

Python II

- Review
- JSON including Pickle and other alternatives
- REST API
- MySQL API (Assumes some SQL knowledge)
- Decorators
- Subprocesses
- Threads
- Multiprocessing
- Test-driven development
- Python Web Server Gateway Interface (WSGI)

THE END