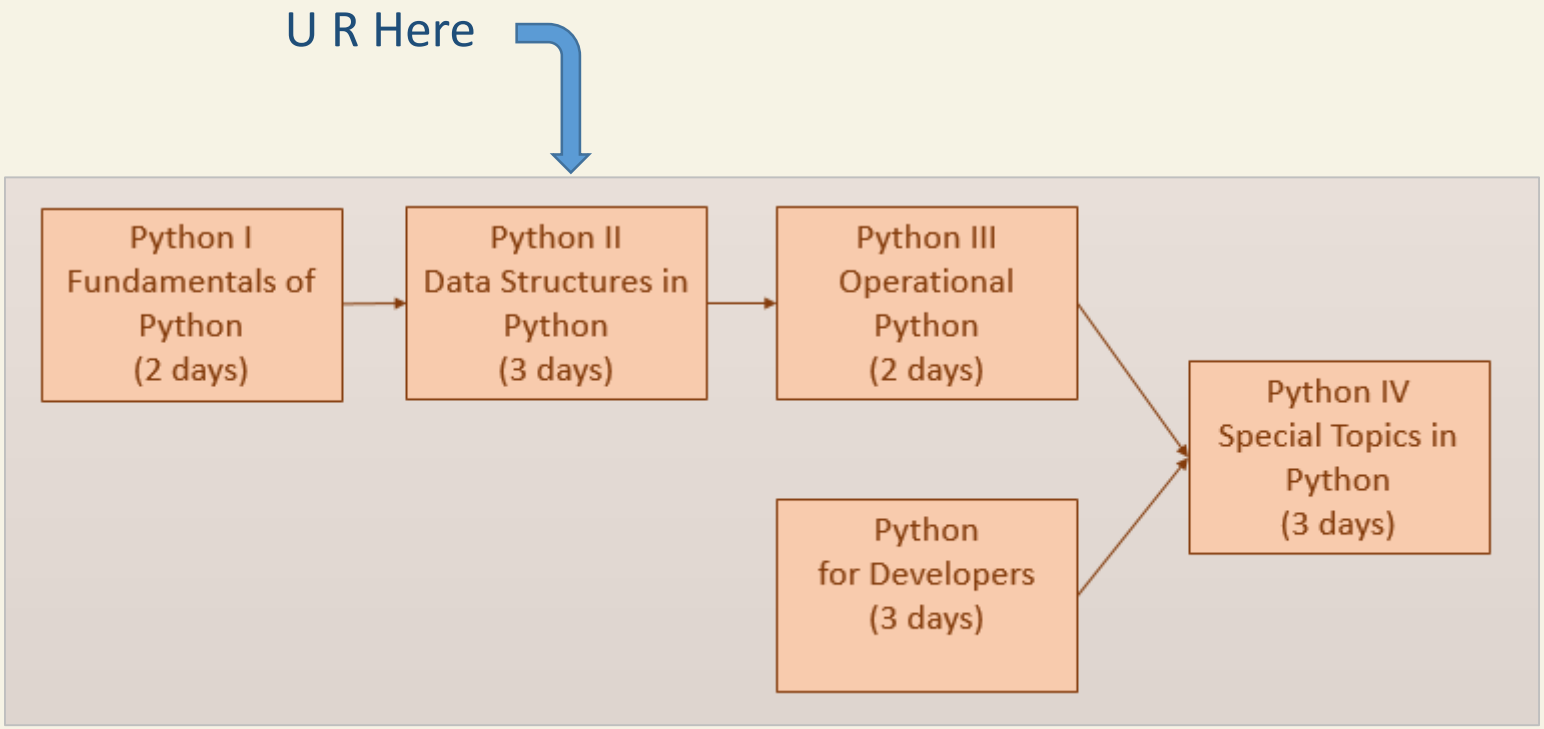
Python II
Welcome to Programming

Rev 07/17/2016

Python Classes



Objective:

- 20% Lecture
- 80% Lab

PAPERWORK

- NLC registration
- RU roster
- Email
  - weastridge@gmail.com
  - Subject: Python
- Think Python, Allen B. Downey
- Python for Informatics, Charles Severance

INTRODUCTIONS

- Instructor
- Students
  - Did you attend Python I?
  - What have you done since then?
  - How do you plan to use Python?

Castle Lab Environment

- ID – student, password - student
- IDLE Install
  - For CentOS: sudo yum install python-tools
  - For Ubuntu et al: sudo apt-get install idle
  - Windows and OS X come with IDLE installed
    - Macs with retina displays may require a larger font size than the default to make underscores show properly.
- CentOS7 at the Castle has IDLE already installed.
- To create an icon for IDLE:
  Drag the file Idle.desktop to the desktop, double-click it and trust it. This is your icon for IDLE.
- Data
  - go to tinyurl.com/py2dataz
  - This file contains documents, samples and data for labs

NLC ENVIRONMENT

- Windows setup:
  - Sign on: ID – rackspace; password – Fall2013 (case sensitive)
  - Open Computer and go to C:\Python27\Lib\idlelib\
  - Right click on idle.pyw and send shortcut to Desktop
  - On the Desktop, double click on the new icon and choose to open with an installed program.
  - In C:\Python27 choose idle.bat in idlelib
  - On the C drive, create two folders: pyprogs and pydata
  - Get the data from tinyurl.com/py2dataz and put it in pydata. This is a zip file.
- Alternative - Boot from USB drive preloaded with Lubuntu and all data and software.
  - NLC computers in CATE 103 and 104 boot from USB as first priority.

Class Structure

- We will use IDLE or Vim and command line. Remote users with Python installed locally will use that environment.
- Outline – Data Structures
  - Review Python I
  - Strings – operations, methods and concepts of iterable and mutable
  - Lists and tuples – operations, methods and concept of tables
  - Dictionaries – concept of quick, direct access to data
  - Sets – concepts of inclusion, exclusion and uniqueness

Python I Review

- What numbering system (base) does the computer use internally?
  - What is the decimal number 54 in this system?
- What is the first thing you do when writing a program?
- When writing a program, what should you always have?
- What is a program failure caused by? What is the solution process called?
- What is source code? Object code? Byte code?
- What are three types of problems you can encounter when running your program?

Python I Review

- Name three possible run-time exceptions.
- Python is a scripted language. Name two more.
- What is an identifier/variable? What two items does it point to?
- What characters can be used to form an variable name?
- Are the variables x and X the same thing?
- In Python 2.7, what is the difference between 7/3 and 7//3? In Python version 3?
- If x and y are strings, is x + y valid? x * y?
- Name several built-in and imported functions you have used.

Python I Review

- What is a Docstring? Where is it most commonly used?
- How is a comment created? Where can they be placed?
- Have you read the first few screens of PEP 8?
- Does a print statement always cause a linefeed?
- What are the five basic data types?
- What is an assignment statement? An expression?
- What formatting characters did we use in Python I?

Python I Review

- How does Python delimit a suite of source code?
- What are some of the comparison operators?
- What logical elements/words are used to make multiple comparisons in a decision process?
- What statement determines whether a suite of code will be executed? How many options does this statement have?
- What two statements cause a suite of code to be executed multiple times? What is the fundamental difference between these two?
- What two statements allow you to prematurely end a given loop? What is the difference between these two?

Python I Review

- What characters can be used to enclose a string?
- What is the pydoc command used for?
- What is the basic purpose of a function? Where in the program is it usually located? Must it be there?
- What is the difference between global and local variables?
  - Where did they occur in Python I?
- What statement traps exceptions? What is a bare except?
- Before an external file is read, what must be done?
- What two ways did we discuss to read a file?

Python I Review

- How did we determine a complete file had been read?
- When we read numeric data from a file or the keyboard, how was it received by our program? What data type was it?
- Did any white-space characters at the end of a record interfere with int() or float() conversions?
- What format allowed us to print the white-space characters such that we could see them?
- When opening a file, how do you determine what operations are permitted?
- If we write to an existing file, what happens to the original data?

Review - Python Help

- help() in the interactive shell. >>>
  - help(int) built-in function
  - help('random') importable module
  - help('random.randint') function in importable module
- At bash prompt: pydoc
  - Uses docstrings from source code
  - pydoc int, pydoc random, pydoc random.randint
  - pydoc –k string: searches for string in the synopsis lines of all available modules
- At powershell prompt: python –m pydoc ---------
- Google!

Review LAB – Dice Roll

- Create a program that calls a function that simulates the rolling of a pair of dice.
- Your main program will deal with the total of the two dice.
- The rules are as follows:
  - On the first roll, a total of 7 or an 11 is an automatic win
  - On the first roll, a total of 2, 3 or 12 is an automatic loss
  - Any other number is called the Point.
  - Keep rolling the dice until one of the following occurs:
    - You roll a 7 which is a loss
    - You roll the Point number again which is a win.
- You start with $100 and bet $10 on each play.
- Print all the rolls and whether you have won or lost on one line.
- Print the funds balance and a request to play again on the next line. A ‘y’ or ‘Y’ means play again. Anything else ends play. A balance of $0 ends play automatically.

Lab Results

```
Beginning Balance = $100
7 You win!
Balance = $110 - Play again? y/n: y
10 7 You lose!
Balance = $100 - Play again? y/n: y
8 6 5 9 8 You win!
Balance = $110 - Play again? y/n: y
11 You win!
Balance = $120 - Play again? y/n: y
8 7 You lose!
Balance = $110 - Play again? y/n: n

Number of plays - 5
Ending Balance = $110
```

Topics – Data Structures

- **Strings** – parsing strings to locate and/or transform the data
- **Lists/Tuples** – all languages have mechanisms for tables. Lists and tuples provide this capability in Python.
- **Sets** - are an unordered collection used for membership testing and eliminating duplicate entries.
- **Dictionaries** – are indexed by *keys and* used for quick lookup and retrieval.
- All of the above structures plus files are iterables which makes them perfect for use in 'for' loops as you will see.

Strings – Basic Operations

- Strings are sequences. As such they support:
  - The len() function.
    - x = ‘himalayas’
    - len(x) produces 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 the ASCII chart)
  - "abc" < "xyz“ (True)
  - "ABC" == "abc“ (False)
  - “abc” < “ABC“ (?) "abc-" > "abc_" (?) "abc1" > "abc" (?)

Strings - ASCII

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Strings – Slicing

- x = ‘himalayas’
- 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
    - x[0] is ‘h’, x[3] is ‘a’, x[6] is ‘y’,
    - 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

Strings – Slicing

- x = ‘himalayas’
- Slices can be negative.
  - x[-1] is ‘s’, x[-3] is ‘y’, x[-9] is ‘h’
  - x[:-1] is ‘himalaya’, x[-6:-1] is ‘alaya’
- Mixed notation - x[2:-2] is ‘malay’
- Steps – x[2:8:2] is ‘mly’ You do not get the end point!
- x[9] is what?

Iteration and Iterables

- ***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.
- To ***iterate*** *over* a sequence (or container) means to visit each element of the sequence, and do some operation for each element.
- In Python, we say that a value is an ***iterable*** when your program can iterate over it. In short, an ***iterable*** is a value that represents a sequence (or container) of one or more values.
- One of the most common uses for an iterable is in a ***for*** statement, where you want to perform some operation on a sequence of values.

Strings – Iteration

- x = ‘himalayas’
- Strings are iterables.
- Traverse a string the hard way:

```
for j in range(len(x)):
    print x[j]
```

- Traverse a string the easy way:

```
for j in x:
    print j
```

- Essentially, j becomes each character in the string.
- Sample – a1StringSlice.jpg

Strings

- What does this function do?

```
def find(word, letter):
    index = 0
    while index < len(word):
        if word[index] == letter:
            return index
        index += 1
    return -1
```

Strings

- What does this do?

```
word = raw_input("enter a word: ")
letter_to_find = raw_input(
    "enter a letter to find in the word: ")
count = 0
for letter in word:
    if letter == letter_to_find:
        count += 1
print "found", count, "occurrences of", letter_to_find
```

LAB 02a

In your data folder, you will find a file containing the text for the book, “Alice in Wonderland.” Read the entire file into memory. Using only the tools we have covered so far, scan this text counting all of the letters regardless of case. Keep a separate count for all occurrences of the letter ‘e’ again regardless of case. At the end, print the total of all letters, the number of e’s and the percent of the total that the e’s comprise. Go to the internet to see if this percent is in line with what you would expect.

You will modify this program in another lab

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.
- Generic example:
  - fin = open(‘path to file/filename’, ‘r’)

    for linein in fin:
        process the record
    statements to execute upon file completion.
  - Sample - a2Read_File2.jpg

LAB 02b

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 precipitation and the precipitation total for the year and print them out. Be sure to account for invalid data.

Strings - Methods

- Methods are just functions acting on an object.
  - In this case, the object is a string
- Remember, functions can be fruitful or void (Result of None)
- All of the string methods (functions) are fruitful.
- 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.
- String "methods“ (See Python Notes for a complete list)

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
- See sample bStringMethods1.jpg

LAB 02a Revisited

In your data folder, you will find a file containing the text for the book, “Alice in Wonderland. Read the entire file into memory. Using only the tools we have covered so far, scan this text counting all of the letters. Keep a separate count for all occurrences of the letter ‘e’.

Review the program you produced for this lab and use the string methods we have covered to simplify the code.

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’

LAB 02c

Using the chr() and ord() built-in functions, print the ASCII characters corresponding to the numbers 32 through 126. Then, from the string module, import the variable printable. If necessary, review the string module through help('string') in the shell or pydoc string from command line. Afterwards, iterate through the “printable” string and print out each entry and its corresponding ordinal. Be sure to use !r (or the older %r) formatting on the string items so you can see the whitespace characters.

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
- There is a screen-scraping exercise assigned as homework at the end of the class.

LAB 02d Alternate

In the book, “Alice in Wonderland” find the words caterpillar and gryphon. Print the location of the first occurrence of each word. Also, print the number of times each word occurs in the book.

If you finish early, determine the location of the last occurrence of each word. There is a method that will do this for you.

Lists

- Python's implementation of tables/arrays.
- Can be multiple dimensions.
  - We will only deal with two at most.
- Advantage – you use one variable name to access/store multiple items.
- Lists are produced using square brackets or the list function.
- Lists can contain items that are all the same type or many different types.
- Like strings, lists are accessed with integer indexes
  - An index can be a literal, a variable or an expression
- 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]

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]

- Sample - dLists1.jpg

Lab 3a – Probability

Plan and execute the following program. Create a function to simulate the rolling of a pair of dice. Call this function 100,000 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. Visually compare your results to the mathematically derived results in the adjacent table.

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 (void)
  - append() vs. insert() vs extend()
  - sort() vs. reverse()
  - remove()
- Produce a return (fruitful)
  - count(), index()
- Both – in-place change and a return
  - pop()
- Sample - eLists2.jpg

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 as you will see later
- my_list.append([4]) # inserts a list within the list
- my_list = my_list.append(4) # wipes out list

Sample - fLists3.jpg (demonstrates iterating through a list)

LAB 03b – Filter, Map Reduce

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

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

- Use the techniques you have learned so far to:
  - 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)
- Use normal loops to accomplish each of these tasks

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.

LAB 03c

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.

LAB 03d

From exercise 10-8 on page 118 of the book, "Think Python":

If there are 23 students in your class, what are the chances that two of you have the same birthday? You can estimate this probability by generating random samples of 23 birthdays and checking for matches.

Write the program such that you run the exercise at least 100 times. At the conclusion of the program print the number of times duplicate dates were detected. As suggested in the book, use the randint function in the random module to generate numbers from 1 to 365 to simulate dates.

If you are using the digital book, the page number is marked as 98. You can find it by typing 120 into your PDF reader.

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

Stacks

- What's a stack? (last in / first out)
- Implementing a stack
  - my_stack = [1, 2, 3]
  - my_stack.append(8) # push with append()
  - top = my_stack.pop() # pop with pop()
- What other ways can you do this operation?

Queues

- What's a queue? (first in / first out)
- Implementing a queue
  - my_queue = [1, 2, 3]
  - my_queue.insert(0,88) # add with insert()
  - last = my_queue.pop() # remove with pop()

LAB 04a

1. Implement a stack using a list data type. Pushing five elements onto the stack. Then, remove each item by using the built-in list method pop(). Print the stack at each change.
2. Implement a queue using a list. Insert 5 elements onto the queue using the insert() method and empty the queue using the pop() method. Print the queue at each change.

Two-Dimensional Lists

- Lists can be many dimensions. We will deal with two. See gLists4.jpg
- How do they sort? Demo with hLists5.jpg in Samples
- List 1

eggs
milk
bread

List 2

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'
- Initialize a two-dimensional list – see samples hLists6 & 7

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

Lab 04b

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

Copying vs Aliasing

- 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 = 1000; y = 1000; x is y
    x += 1; y += 1; x == y; x is y
    immutable example
  - x = [1, 2]; y = x; y.append(3)
    x == y; x is y; print x, y
    mutable example

Debugging Pearls Of Wisdom

- Most list methods return None.
- Watch for in-place modification.
- Lots of legitimate ways to change a list:
  - add: append(), insert(), extend()
  - delete: pop(), remove(), del, slice
- Make copies: help stamp out list aliasing
  - nu_list = existing_list creates an alias
  - nu_list = list(existing_list) creates a new list*
  - nu_list = existing_list[:] creates a new list*

* This works only on one-dimensional lists.

Tuples Are Sequences Too

- 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 as return values.
- Tuples show up in lots of places.

How Are Tuples Used?

- Functions can return only a single object
  - But a tuple is an object, so for > 1 return object, use a tuple
- 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?

Comparing Tuples

- Relational operators work as expected
- (1, 2, 3) < (1, -2, 3) ?
- What is 1, 2, 3 < 1, -2, 3 ?
- Sort an immutable?
  - Use the sorted function
  - try x = (1, -42, 138, 18); y = sorted(x)
  - What is the result of the above operation?
  - What data type is y?
  - z = reversed(y) What is z?

Variable Parameter Collectors

- def fn(*varname) varname is usually args
- *varname parameter receives all excess positional arguments
  - “Packs” them into a tuple
- There is a similar operator for keyword parameters.
  - Good explanation here, but it gets complex.
- 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.
  - Or: if isinstance(x, (type1, type2, …)) for multiple types
  - Do not put the type in quotes.

LAB 05a

There is a program in your data file called temp_convert.py. It has a function that converts a Fahrenheit temperature to centigrade. Change this program 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 it. Each parameter should be either int or float. Reject all others. Test with invalid data.

Example function call:

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

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

Unpacking a Collection

- Hypothetical situation:
  A function is expecting a variable number of positional parameters, but you have the data in a collection such as a list or tuple.
  - In the previous lab, passing the list or tuple will be received as one item.
- Using the * in front of the name in the function call unpacks the collection.
- Example:
  x = [1, 2, 3, 4]
  func_name(x) # sends one item (a list) to the function.
  func_name(*x) # sends four integers to the function
  func_name(*x[1:]) # what does this do?

Command-Line Parameters

- You can send parameters to your program from the command line.
- This is done through the argv variable in the sys module.
- Do the following:
  - Create the following two-line program:
    from sys import argv
    print type(argv), argv
  - Execute your program from command line twice:
    python progname.py
    python progname.py 2016 04 13
  - What data type is argv? What is always the first item in argv?

LAB 05b

Create a copy of the program from Lab 05a and change the main program to accept a variable number of parameters from the command line. Send those parameters to your function which is still accepting a variable number of inputs. As before, have the function parse/test the arguments using different tools as necessary. Print all results.

Example command with parameters:

```
python lab_05.py 72 -10.5 a 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 : e.g., numbers, strings, tuples.
- Keys are hashed for fast lookups. (See Python Notes for a discussion of hashing)

Dictionaries

- Examples:
  dict_01 = dict() creates an empty dictionary
  dict_02 = {'sun': 1, 'mon': 2, 'tue': 3, and so on}
- General Information
  - Order of items unknown
  - Accessing - dict_02['mon'] returns 2
  - KeyError access exception
  - len() tells you the number of key:value pairs
  - *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
- In “Think Python” read about dictionaries as counters (page 123 – hard copy or page 102 in the pdf version. In your PDF reader enter page 124)

Dictionaries as Iterables

- *for* loop uses the dict keys to iterate by default
- Value is accessed with each key as in the sample.

Formatting Review

- Formatting sequences:
  - {#:w,.nL}, where:
    - # - is the
    - w specifies the width of the field
    - .n specifies the number of decimal places or the number of digits for an integer
    - L is r, s, d or f. There are more.
- The format() built-in function adds commas.
  - x = 1234567.891
  - print ‘Company profit was $%s last month’ % (format(x, ‘,.2f’))

    Company profit was $1,234,567.89 last month

Formatting Data into Strings

General example:

‘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

Abbreviated general format: {[seq#] ":" [width] [","] ["." prec] [type]}

Examples:

a = 12

b = 17.426

x = 'Some text {} more text {}'.format(a, b)

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

x = 'Some text {0} more text {1}'.format(a, b)

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

x = 'Some text {1} more text {0}'.format(a, b)

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

x = 'Some text {0} more text {1:.2f}'.format(a, b)

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

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

Formatting Data into Strings

Abbreviated general format: {[seq#] ":" [width] [","] ["." prec] [type]}

General examples:

a = 1234567.889

x = 'I would like to have ${}'.format(a)

Result stored as x – 'I would like to have $1234567.889'

x = 'I would like to have ${0:,.2f}'.format(a)

Result stored as x – 'I would like to have $1,234,567.89 ' (Add separators – note rounding)

x = 'I would like to have ${0:15,.2f}'.format(a, b)

Result stored as x – 'I would like to have $ 1,234,567.89'

When the width specified is larger than necessary to accommodate the result, numbers are right justified and everything else is left justified.

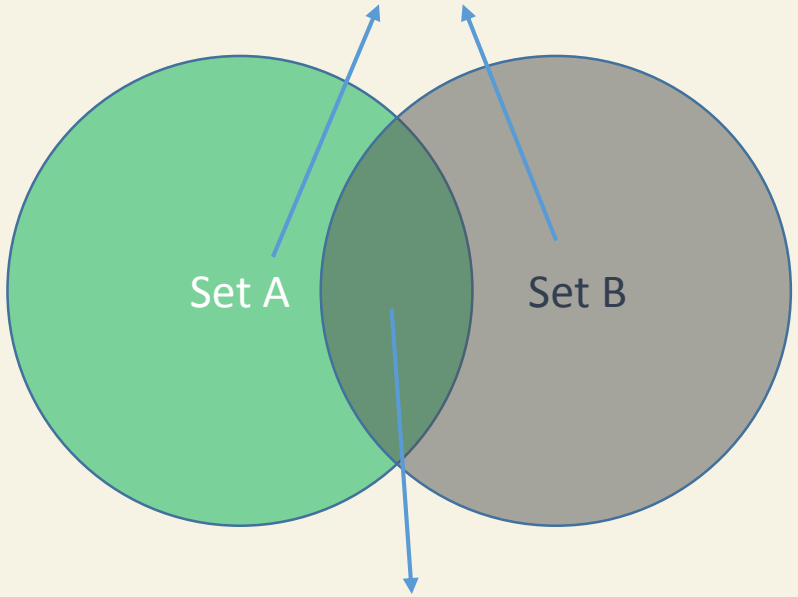
LAB 06a

Read the book, "Alice in Wonderland" into memory. Create a dictionary counting all the printable characters excluding whitespace. Be sure not to count upper- and lower-case letters separately. Creating the dictionary is the most important part of this lab. When done, print the top 30 most frequently occurring characters along with the number of occurrences.

If you have time, print five character/occurrences combinations per line. Make sure all the elements of the printed lines form neat columns.

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.

Sets

- Sets are the final data type we will study
- 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
  - union: sum of both sets with duplicates removed
  - intersection: in both sets
- See the set methods in Python Notes
- Sets are unordered.
- As with most iterables, the len() built-in function is operable.
- See sample iSets.jpg

LAB 06b

In your data file is a program named servercheck.py. It reads two files (servers and updates) and converts the contents into two sets. The updates are not always correct. You will find all of the set operations/methods in Python Notes. Using just these operations/methods, your job is as follows:

1. Determine whether the list of updates exists in the master server list. Print a message indicting whether or not this is true.
2. If it is not true (and you know it isn't), create a new set containing the update items that are NOT in the master server set. Print the number and names of the unmatched servers.
3. Create a new server set that excludes the valid updates.
4. Print the number of items in the original server set and the new server set as well as the number of valid updates.
5. Write the contents of the new server set to an external file using the writelines file method.

Strings (Again)

- In the last lab, we ignored the newline (\n) characters after each server name.
- We can change all of that with two string methods: splitlines and join.
- Splitlines creates a list of lines broken on line boundaries. (See Python Notes for details)
- Join concatenates all of the entries of a collection (e.g., a list). It separates them using the string upon which the method is acting:
  ex: somestring.join(somecollection)
- What does this produce?
  my_lst = 'first\nsecond\nthird\nfinal\n'.splitlines()
  prline = '\n'.join(my_lst)
- Do the above operations in the shell. Display each result.

LAB 6c

Change the program you developed in the last lab to remove the newline characters from the data you read in. Then you have to put them back in the data you write. In this case, how will you read the server data? How will you write the new server file? Be sure to answer these questions before you write any code. When finished, you should be able to open the new server file with any text editor and have it display one server per line.

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()
```

- Remember, the string module has useful variables. (e.g., punctuation)
- Samples - kStringMethods2.jpg

LAB 08

Use the split() method to process the following files: gdp.txt and split.txt. Examine each file and determine how best to separate the various elements to accomplish the assigned task.

In the first file, each record has three elements: country name, gross domestic product (GDP) in millions of dollars, and total population. Your job is to calculate the GDP per person for each country and print out country and GDP/person in descending order of per-capita GDP. Format the results for a professional look.

In the second file, determine how many words there are in the file and how many of those words are unique. Then print out each word in ascending order. Be sure to change every alphabetic character to one case and remove/replace all punctuation.

Optional Lab

- Read the tmpprecip.dat file and use a dictionary to accumulate the data necessary to report the following for each year:
  - Total rainfall (precipitation)
  - Maximum high temperature
  - Minimum high temperature
  - Average high temperature
- Use a separate dictionary to report the average rainfall by month.
- Create well-formatted reports from each dictionary
- The format of the data is the same as it was in Lab 2b. The data goes from 1/1/1900 to 12/31/2012.

Next Steps

Before going on to Python III:

- Review chapters 1 – 10 in, “Python for Informatics.” Make sure you cover the vocabulary and exercises as well. This is basic foundational material. You should be reasonably comfortable with it.
- Do all of the exercises from Learn Python the Hard Way (LPTHW) through #39
- Do the optional lab if we didn’t have time in class. Usually, we don't have time. It requires you to use a list as the value portion of a dictionary.
- In the book, “Think Python," complete exercise 8-12 on page 96. Also, make sure you have done the Take-Home Lab in the slide following this one. I will send you the completed labs upon request.
- Understand all of the labs from Python II.

Next Steps

Do the following lab:

In the data from Python II find, “alice_in_wonderland.dat.” You will also find a file labeled, “words.txt.” The latter file contains over 100,000 English-language words. Your job is to perform the following:

Create a dictionary for counting using the entries from words.txt as the keys.

Parse the text in alice_in_wonderland.data isolating each word. This requires removing/replacing all punctuation and using the split method. Make sure the words in the book are all lower case.

Find each word from the book in your dictionary and increase the count for that word by one.

If a word is not found in the dictionary, place it in a list with other unfound words.

When you have processed the entire book, determine the percentage of words in the dictionary that were used in the book and which word was used the most.

Remove the duplicates from the list of unfound words, sort it and print it.

Don't try too hard to make this perfect. It won't happen!

Next Steps

The output from your program should look something like the following:

Words in dictionary - 113,814

Words in book - 26,694

Percentage of dictionary words used in the book is 2.19%

The word "the" was the most frequently used at 1,644 times

Words not in the dictionary:

30 ada alice alices alternately

ann arrum australia barrowful beauti

c canterbury carroll cartwheels chatte

cheshire christmas couldnt d delightful

didnt dinah dinahll dinahs dinn

doesnt dont dormouses duchesss e

edgar edwin elses elsie england

esq est everythings favourite footmans

france ful hadnt hasnt havent

….. and so on

The End