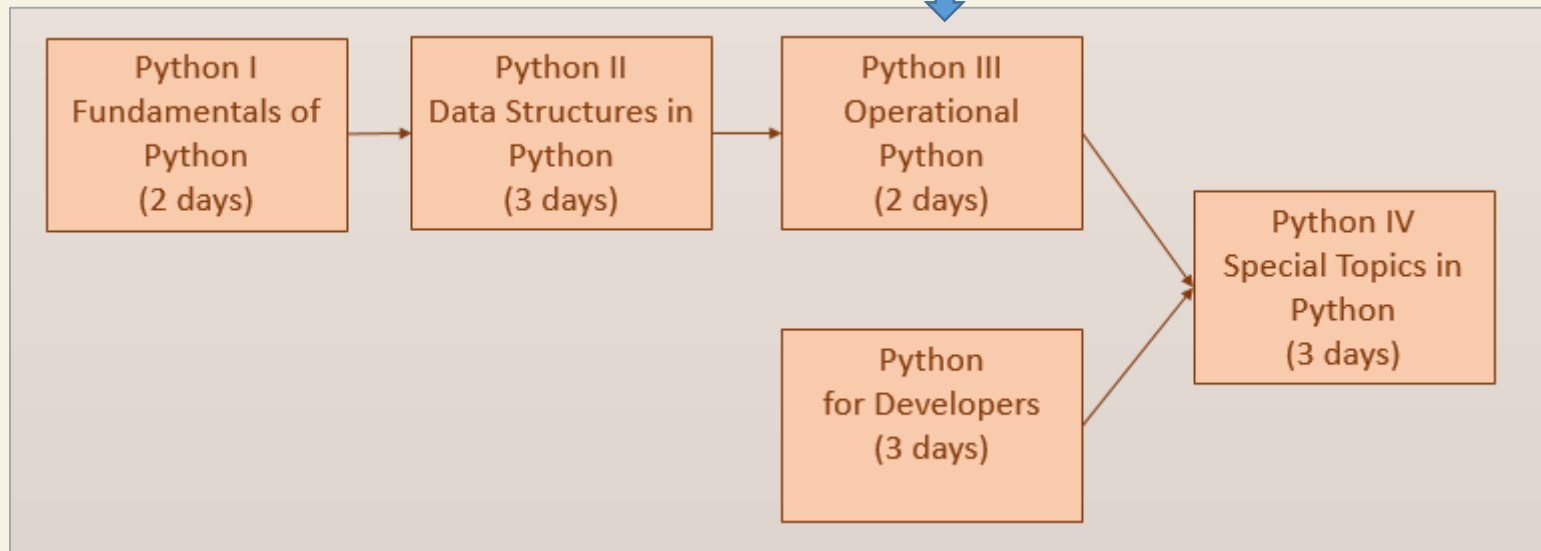



PYTHON III



Python Classes

U R Here



Objective:

- 20% Lecture
- 80% Lab
- Use Python V3

INTRODUCTIONS

- Instructor
- Students
 - What is your current job
 - Python-wise, what have you done since Python II?
 - What is your objective/goal in taking this class?

PAPERWORK

- NLC registration
- RU roster
- Questions? – send me a message
 - weastridge@gmail.com
 - Subject: 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 idle3`
 - 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 `Idle3.desktop` to the desktop, double-click it and trust it. This is your icon for IDLE.
- Data
 - go to rebrand.ly/py3data
 - This file contains documents, samples and data for labs

NLC ENVIRONMENT

- Windows setup:
 - Sign on: ID – rackspace; password – Qtr and Year (e.g., Fall2014 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 pythonw.exe
 - On the C drive, create two folders: pyprogs and pydata
 - Get the data from tinyurl.com/py1data 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.

Resources

- You should already have two books in PDF format:
 - Think Python
 - Python for Everybody
 - These are available at rebrand.ly/py2data
- 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.
- Completed Labs – a collection of completed labs to be sent at the conclusion of the class.

The Zen of Python, by [Tim Peters](#):

- 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!

Class Structure

- We will use IDLE or Vim and command line. Remote users with Python installed locally will use that environment.
- Outline
 - Review
 - Python version 3 changes
 - Clarifying/Expanding – translate, with, sorting, copying
 - Comprehensions, lambda functions, generators and recursion
 - Classes, namespaces, scoping and importing
 - os and sys modules and debugging tools
 - Using the 2to3 converter.

Review

- Which of the following are sequences? Iterables? strings, lists, tuples, dictionaries, sets, files
- How many bits does it take to define an ASCII character?
- What does `ord('a')` produce? What does the `chr()` function do?
- Name four string methods we used in the previous class and their purpose.
- What operator concatenates two strings? Replicates a string?
- What function provides the number of characters in a string?
- How do you create a new string without the last character of the original string? Without the first character? Reversed?
- If the variable `x` is a string, what does `x.split(',')` do? How about `x.split()`?

Review

- Why is the following code considered sloppy?
`updates = set(open('c:/pydata/serverupdates.txt', 'r'))`
- What are the two basic ways to create a list?
- What data types can a list contain? Must all items in a list be of the same type?
- In a two-dimensional list named `my_lst`, how do I access the third entry in the fifth row?
- What does `x` contain after executing the following: `x = my_lst.sort()`?
- What is an alias? How did we learn to prevent them?
- How would I remove the last item in a list and save it in variable `y`?
- What is the difference between the `sort` method and the `sorted` function?

Review

- What is a tuple and how does it differ from a list?
- Are there any methods that work on a tuple?
- Where did we see tuples in previous classes? How were they used?
- What are the two basic purposes of sets.
- What kind of data can a set contain?
- What will this set contain: `x = set([12, 3, 2, 12, 5, 2, 3, 12, 5])`?
- Name two methods used with sets and what they accomplish.
- What data type is `x` if `x = {1, 2, 3}`? If `x = {}`?

Review

- What is the structure and purpose of a dictionary?
- What does the len function produce when applied to a dictionary?
The in operator?
- Why would you get a KeyError exception accessing a dictionary?
What method will avoid this error?
- What type of data can reside in the key portion of a dictionary? In the value portion?
- For the dictionary dc1, what does dc1.keys() produce?
dc1.values()? dc1.items()?
- Have you discovered an alternative to the following comparison:
x >= 50 and x <= 100?

Lab 01 - Review

- In your data you will find the file labeled, “alice_in_wonderland.dat.” You will also find a file labeled, “words.txt.” The latter file contains over 113,000 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.
 - 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!
- Hint: Remove apostrophes and replace all other punctuation with spaces.

with Statement

- This statement is a "context manager" which can handle a wide variety of situations.
- The simplest of these involves files.
 - Garbage collection/cleanup routines close files most of the time.
 - Good programming technique requires that you don't depend on it.
- What does this code do?
 - try:
 filein = open('some file', 'r')
 do a bunch of other stuff with the file
finally:
 filein.close()

with Statement

- The **with** statement makes this simpler
 - with open('some file', 'r') as filein:
do a bunch of other stuff with the file
 - The above code does the same as the try/finally
 - As usual, everything you do with the file has to be indented.
 - At the end of the suite of code, the file is closed automatically
- Continuations (from PEP 8)

```
with open('/path/to/some/file') as file_1, \
    open('/path/to/another/file', 'w') as file_2:
    file_2.write(file_1.read())
```


Lab 02

Go back and change the last lab to open both files in a with statement. Remember, this precludes the need to close the file. My version of the last lab is available online.

fromkeys Dictionary Method

- The fromkeys method can be used to easily and efficiently initialize a dictionary. It is our first static method.
- A static method does not use the object it is attached to. Python still forces you to attach it to the correct data type.
- General format: dict.fromkeys(seq[, init]) init defaults to None
- Comparison: (Do these in the shell)

```
x = 'abcd'
```

```
y = ['wrd1', 'wrd2', 'wrd3', 'wrd4']
```

```
z1 = dict.fromkeys(y); z2 = dict.fromkeys(y, 0) # The dictionary dict is
```

```
z1 = {}.fromkeys(y); z2 = {}.fromkeys(y, 0) # irrelevant as is {}
```

```
z1 = dict.fromkeys(x, -1) # works the same with a string
```

translate Method

- An easier way to do multiple conversions/deletions in a string is with `translate`.
- `Translate` takes one required argument. It is a dictionary containing each character to be replaced or deleted.
 - You use the `maketrans` static method to create this dictionary.
- `Maketrans` produces the dictionary required by `translate`. It has two required arguments and an optional argument:
 - The first argument contains the string of characters you wish to modify.
 - The second contains the corresponding replacement characters.
 - The optional third argument contains the characters you want to delete.
 - `Maketrans` is now a built-in static method. No import required.

translate Examples

```
x = "Don't worry-be happy!"
```

```
temp = str.maketrans('y!', 'i?', "'")
```

```
y = x.translate(temp) # or
```

```
y = x.translate(str.maketrans('y!', 'i?', "'"))
```

y now contains:

```
"Dont worri-be happi?"
```

```
z = x.translate(str.maketrans("'", '"', '-?'))
```

z now contains:

```
"Dont worribe happi"
```

Lab 03

Go back to the last lab and use the `translate` method to deal with the replacement or removal of punctuation. Also, create the dictionary from `words.txt` by using the `fromkeys` method. My version of the last lab is available online.

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.
- Given `x = [12, 18, 200, 5, 1, 10]`, what do these produce?
 - `y = sorted(x); z = reversed(y)` – data type and contents
 - Given what we have covered in Python II, how can we combine the above operations?
- Review `b0sort.jpg` in Samples

Sorting

- Both the sort method and the sorted function have more options:
 - `key=` and `reverse=`
- `reverse=` allows reverse to be incorporated into the sort
 - Ex: `reverse = True` – reverses an ascending sort
- `key=` allows the passing of a function along with a method if desired.
 - Ex: `key = str.lower` – sorts as though all text data is lower case.
- Using itemgetter "function" in the operator module to specify which item(s) to sort on.
 - See `b1Sort.jpg` and `b2Sort.jpg` in Samples

Lab 04

Modify the results of the Lab 03 (available online) to print just the top 20 words along with their respective use counts. Print them in descending order by use count. Do the sorting in one statement using the capabilities we have just reviewed. It isn't necessary to print all the data we have printed previously. The objective here is to ensure you can sort using the capabilities we just reviewed. Also remember, you must unload the dictionary in order to sort the contents.

Copying Mutable Objects

- Copying lists and dictionaries is NOT straightforward.
- Aliases are very easy to create unintentionally
- `x = [1, 2, 3, 4]; y = list(x)`
 - We have just created a shallow copy
 - Only the first level is unique
 - That works fine with this example
- Successive levels beyond the first will be aliased.
- The `deepcopy` function in the `copy` module helps with this problem.
- `Deepcopy` works with more object types. We are only using it for lists and dictionaries here.

Copying Mutables

- List examples:
- Review d1Deepcopy.jpg in Samples
- Review d2Alias.jpg and d2Aliasalt.jpg in Samples
 - Be very careful in initializing lists
 - Deepcopy will not help you here
- Dictionaries with mutable values are vulnerable to aliasing.
- If your dictionary contains lists and/or other dictionaries:
 - Use deepcopy to get an independent copy

Shebang and Encoding

- The shebang allows you to execute the program as a script
 - `#!/usr/bin/env python`
 - `#!/usr/bin/python`
- Use `chmod` to make them executable
- On Windows, simply open with `python.exe`. The shebang is good if you want to be Linux-compatible.
- Python2 uses ASCII by default. Python3 uses Unicode.
- If your Python 2 code goes beyond ASCII use:
 - `# -*- coding: utf-8 -*-` or, `# -*- coding: latin-1 -*-`
 - Refer to Python Notes for encoding articles
 - See `ucode.py` and `ulatin_print.py` in DemoProgs

Review – Filter and Map

This loop selects only the even numbers to be included in a new list
This is an example of filtering.

```
y = []  
for i in range(2,20,3) :  
    if i % 2 == 0:  
        y.append(i)  
print y
```

This loop creates a new list containing the square of each entry in
list x. This is an example of mapping.

```
y = []  
for i in range(2,20,3) :  
    y.append(i**2)
```

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 for x in range(10) if x % 2 == 0]
result - [0, 2, 4, 6, 8]

List Comprehensions

- General Format:

```
[expression for expr1 in iterable1 (if condition1)
      for expr2 in iterable2 (if condition1) ... ]
```

```
y = []
```

```
for i in range(2,20,3):
```

```
    y.append(i**2)
```

}

```
y = [i**2 for i in range(2,20,3)]
```

```
y = []
```

```
for i in range(2,20,3):
```

```
    if i % 2 == 0:
```

```
        y.append(i)
```

}

```
y = [i for i in range(2,20,3) if i % 2 == 0]
```

- Note – review iList8.jpg and jDictionary2.jpg in Samples

Dictionary and Set Comprehensions

- As you have seen with lists, these:
 - are a shorthand way of initializing these data structures.
 - allow control of the number of elements, and built-in filtering all in one operation.
 - The alternative is to use for loops which are sometimes more readable and sometimes not.
- Examples (can you guess which is which?):
 - `y = {x**2 for x in range(5)}`
 - `y = {(x, x**2) for x in range(5)}`
 - `z = {x: chr(x+65) for x in range(20)}`

Lab 06

Using `range(-40, 120, 10)` as input, create three comprehensions. Assume the values in the range represent Fahrenheit temperatures. The comprehensions should calculate the corresponding Centigrade temperature. ($\text{Centigrade} = 5.0/9.0 * (\text{Fahrenheit} - 32)$)

- Create a list and a set using comprehensions. Both the list and the set should contain tuples which contain each Fahrenheit/Centigrade pair. Exclude the entries for Fahrenheit temperatures zero and 50.
- Create a dictionary comprehension with the Fahrenheit temperatures as keys and Centigrade temperatures as values. Use the same exclusions as above.

Functions

- Functions are objects just like everything else in Python
 - They can be contained in / pointed to by another variable
- Also, functions come in several flavors:
 - Lambda functions – unnamed one liners
 - Generators – produce results as needed
 - Recursive functions – call themselves
- We will study all three of these function types

Function Basics

```
def convert(f_temp):
```

```
    c_temp = 5.0 / 9.0 * (f_temp - 32)
```

```
    return c_temp
```

f_to_c = convert # without the parens, this just assigns the function

```
for temp in range(-40, 120, 10):
```

```
    ftemp = float(temp)
```

```
    if ftemp == 0 or ftemp == 50:
```

```
        continue
```

```
    ctemp = f_to_c(ftemp)
```

```
    print('%0.1f degrees Fahrenheit is %0.1f degrees Centigrade' % (
        ftemp, ctemp))
```

Lambda Functions

- Lambda functions are anonymous, one-line functions composed of an expression and often assigned to variables.
- They are used to avoid littering your program with small functions.
- A lambda function can take multiple arguments and produce one result.
- General syntax
 - `lambda input variable(s): expression to create a return`
- Examples:
`a1 = lambda x : x ** 2 # one input - x`
`a1(5)` produces 25
`f = lambda x, y : x + y # two inputs – x and y`
`f(14, 12)` produces 26
- Run these examples in the Python shell.
Then try `f = lambda x, y : (x + y, x * y)`
What is returned for `f(14, 12)`?

Lab 07 - Lambda Functions

Take the program `tempconversion.py` in your data folder 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]` # map and filter produce iterators in Python 3.x
 - `x = map(lambda z : z **2, y); list(x)`
 - `x = filter(lambda z : z % 2 == 0, y) ; list(x)`

Another Lambda Example

- Sorting alternative

```
x = [(0, -1, 3), (-1, 0, 4), (2, -6, -5), (1, 3, -2), (3, 2, 1)]
```

- How do I sort on the absolute value of the second element?

```
sorted(x, key=lambda y: abs(y[1]))
```

 Result:

```
[(-1, 0, 4), (0, -1, 3), (3, 2, 1), (1, 3, -2), (-2, -6, -5)]
```

- Lambdas are used extensively in GUI's (Tkinter, wxPython)
- See `sortkey.py` and `sortrand.py` in `DemoProgs`

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 have seen, 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 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. It also handles the `StopIteration` condition.

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)
- This type of function is called a generator which is a type of iterator
- 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 iterators that don’t pre-generate the results until needed.
- Called with `next()` just like any iterator.
- Saves memory: does not pre-build returned object
 - Ex. `sorted(iterable)` vs `reversed(iterable)`
- See `generator_test.py` in DemoProgs

Lab 08

Write a generator that returns an increasing integer if the integer is even or the string "odd" if the integer is odd. Choose arbitrary starting and stopping points (e.g., 4 and 15). When the numbers are exhausted, let the function end without a yield statement.

Test your generator by using a `*for*` statement and then again by calling `next()` and testing for `StopIteration` manually. Print whatever is returned from the generator in both cases.

Lab 09

Use the starter program called "gambler" in your data folder. Read the docstring for the proposed generator and do the following:

- Get the generator to create the card deck, shuffle it and deal out cards as requested. Use the code provided.
- In the main program:
 - For each card dealt, divide the number such that you have a number for the rank of the card (2 – Ace) and a number for the suit
 - Use the two lists provided to print out the rank and suit of each card. (e.g., Eight of Clubs, King of Hearts)
 - Once the program is working, use it to print out the entire deck.
 - Decide how you want to handle the next and StopIteration situations.

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)

Recursion

- Python supports recursive functions
 - Functions that call themselves
- Recursion is a somewhat advanced topic and not used often
- Why bother to study it?
 - It helps you to understand the abstract nature of Python.
 - It allows programs to traverse arbitrarily shaped structures.
 - Used alternatively to loops and iteration – usually unnecessarily.
- Review `recursive_blastoff.py` and `irregular_sum.py` in DemoProgs
 - Taken from the Think Python and Learning Python books.

Lab 11

Write a program to calculate a factorial recursively. In mathematics, the factorial of a positive integer n is denoted by $n!$ and is the product of all positive integers less than or equal to n . For example, $4! = 4 * 3 * 2 * 1$ and $5! = 5 * 4 * 3 * 2 * 1$ and so on

Your program should receive an integer from the keyboard and use a recursive function to resolve it. The table to the right provides some correct answers for checking your results.

0	1
1	1
2	2
3	6
4	24
5	120
6	720
7	5 040
8	40 320
9	362 880
10	3 628 800
11	39 916 800
12	479 001 600
13	6 227 020 800
14	87 178 291 200
15	1 307 674 368 000
16	20 922 789 888 000
17	355 687 428 096 000
18	6 402 373 705 728 000
19	121 645 100 408 832 000
20	2 432 902 008 176 640 000

Python Classes

- Classes are Python's main object-oriented programming (OOP) tool.
- At their most basic, classes are another way to modularize your code for reuse.
- Classes themselves have no use until they are used in what is called an instance.
 - They are simply a template or a blueprint.
- A major advantage of classes is inheritance; a mechanism of code customization and reuse.
- Classes tend to be used by people doing long-term development (more strategically oriented)

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 and accessed through dot notation.

Class Example (class.py in your data)

```
class BankAccount(object): # Top tier class – "object" is optional
    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 doing the calling. (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.).
- The name `self` is not special to the code, it's just another object.

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.
- Reaching "inside" an instance is generally not a good idea. Normally, you depend on a method for access.
- Usually, you want something more complex done when you print.
 - In this case, I would print the "a" instance
 - But, "print(a)" results in nonsense
- This is where we use Python's Magic Methods.

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.
```

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(2500)
```

```
print(a)
```

Class Example

- How do I make comparisons?

if a == b: Does not work. Why not?

In 2.7 the default is to use `__cmp__` The standard `__cmp__` will not do the comparison correctly.

In 3.x the default is to use `__eq__` The standard `__eq__` will not do the comparison correctly.

if a.balance == b.balance: Will do the desired comparison.

- Often we want to do more complex comparisons.
- These are best done in methods, so everyone using the class can take advantage of them
 - Avoids code replication in main programs

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.

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 13 - Class Example

- When you compare, the default version of `__eq__` is called automatically and it will blindly compare two instances which will never be equal.
- To override this result, 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 methods of the class. Class variables aren't used as frequently as instance variables. Class variables are not inherited.

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 14

- 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.
- If you execute from command line, you may get an error message after the program has run successfully. We will discuss.

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
```

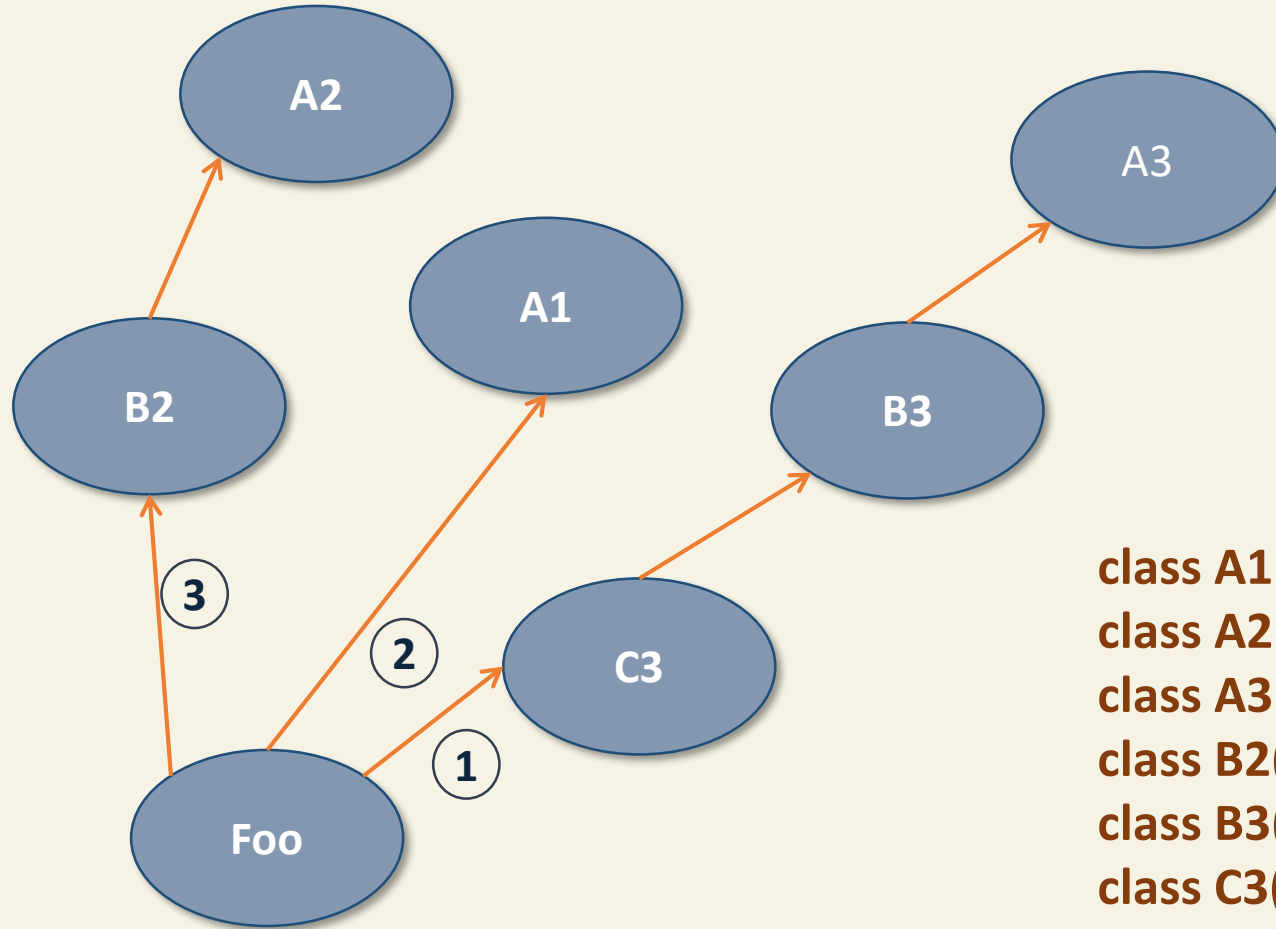
- Class B has inherited all the methods and instance variables of class A. In addition, Class B replaces mthd2 with its own version of that method.

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 15

- 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. For now, don't check to see that the initial deposit meets the minimum requirement.
 - 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

Class Review

- Classes are simply prototypes or models to be used in the creation of an instance object.
- The act of creating an instance is called instantiation because we like five-syllable words. (e. g.; polymorphism, encapsulation)
- Each instance of a class has it's own instance variables that are preserved in the latest state.
- One class can inherit instance variables and methods from another class. Class variables are never inherited.
- An inherited method can be overridden to get a different result.
- The "magic methods" are simply built-in methods that are called automatically in given circumstances, and they can be overridden. (e.g.; `__str__()`, `__le__()`, and so on)

raise Statement

- The **raise** statement is used when code wants some attention because of an event. The attention-getting device is an Exception object
- Below is a generic raise statement:
 - raise [Exception [(0 or more args)]] – where args is usually one string
- In the Shell enter the following and note all results:

```
raise
raise ValueError
raise ValueError('An error occurred')
raise ValueError('An error occurred', 1000)
```
- If a problem is discovered during an instantiation (`__init__`), you can cancel the instantiation by raising an exception.
- See `raise1.py` and `raise3.py` in `DemoProgs`

Lab 16

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.

Namespaces and Scoping

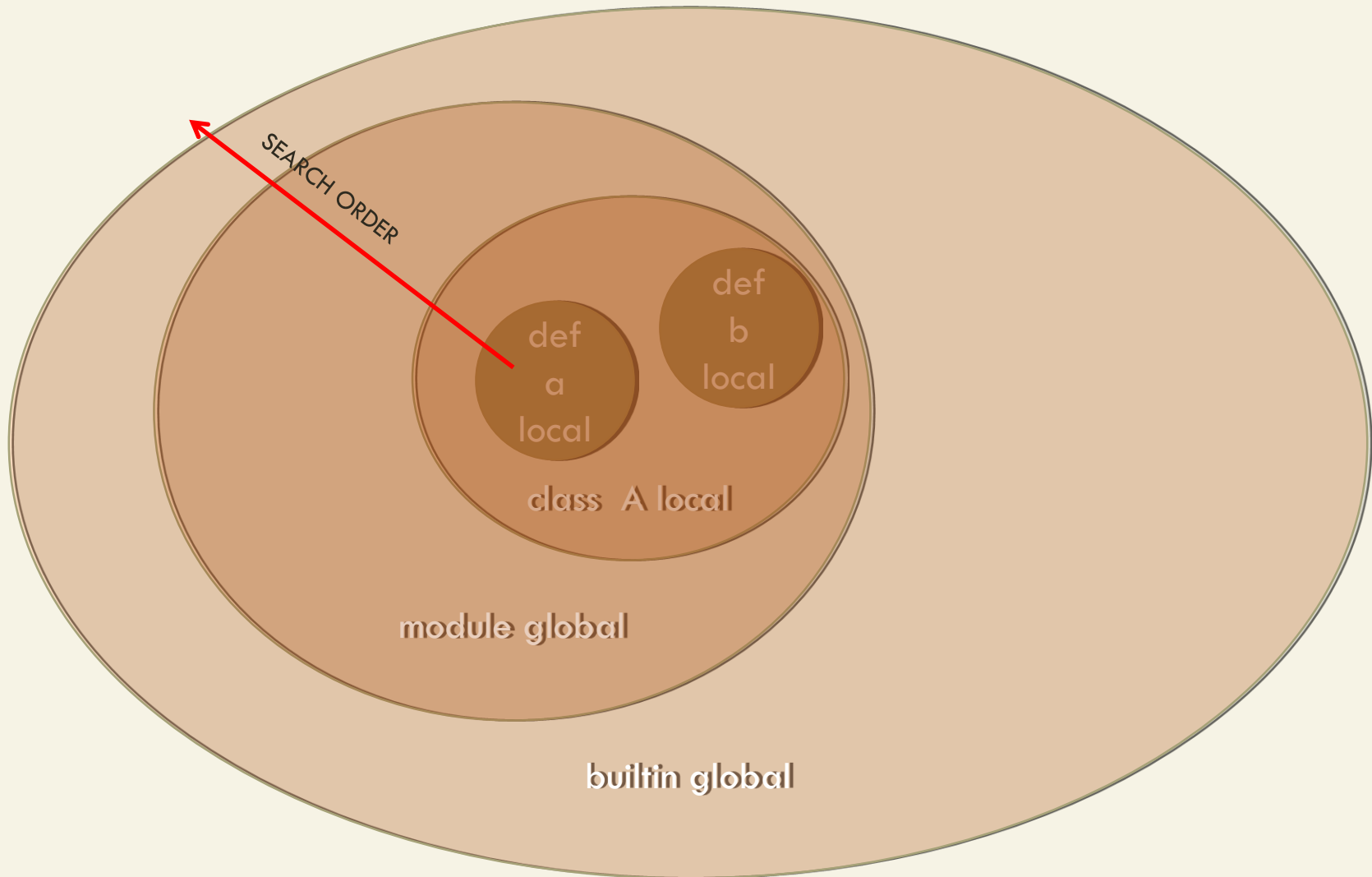
- Everything in a Python program is an “object”.
- Objects have state and behavior.
 - Python stores an object's state in a variable. (x = 'abcde' : a string)
 - It exposes an objects behavior through methods. (x.isalpha() = True)
- **Block:** any Python code executed as a unit e.g. a module, a function, a class or a script.
- 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, classes/instances and modules are not known globally

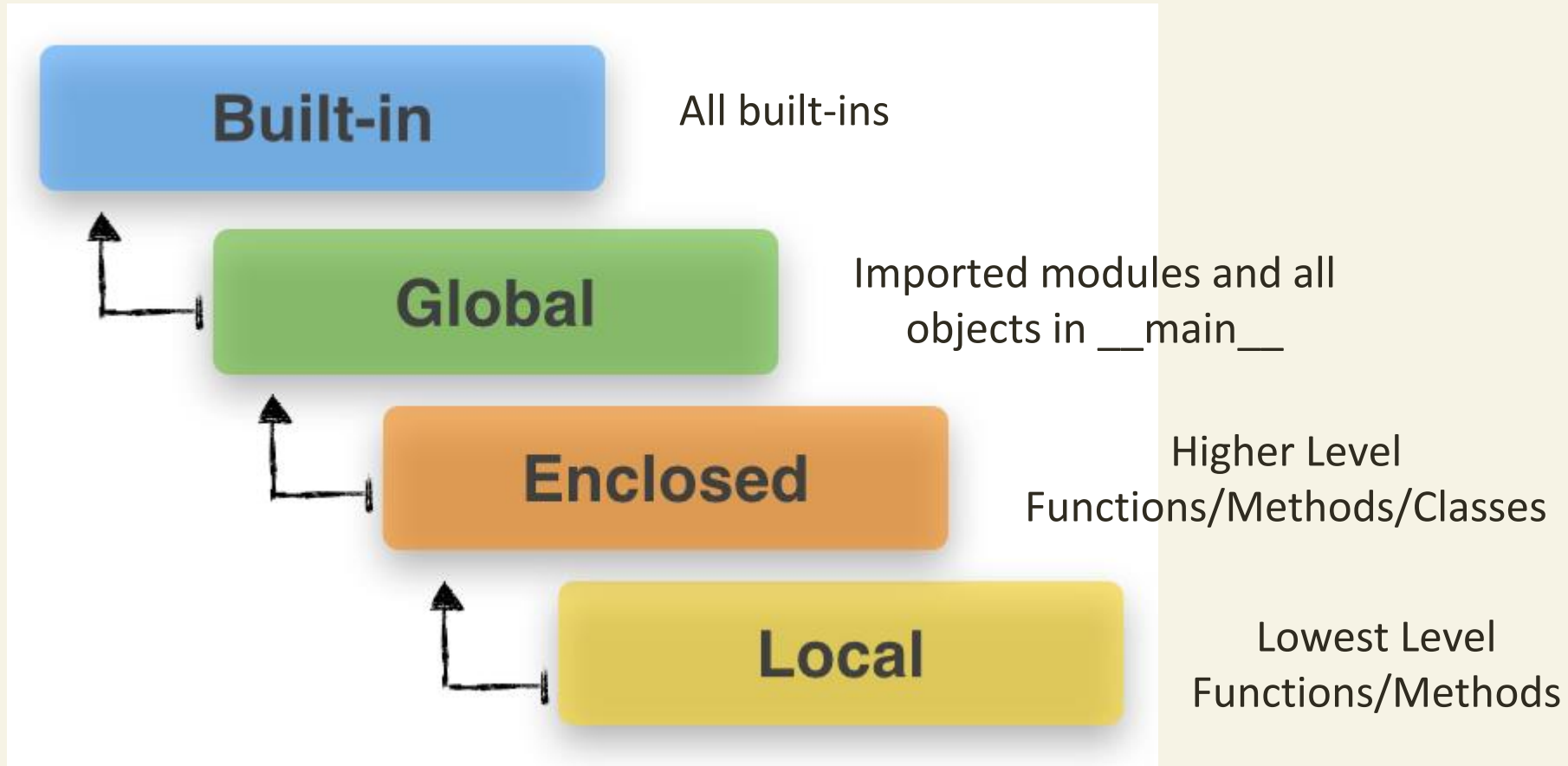
Namespace Diagram

There is a local namespace for each code block.



Namespace Hierarchy

Another way of looking at it



In DemoProgs, run `lcl_glbl.py` and `lcl_glbl_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. All classes and class instances also have their own namespaces as do imported modules.
- 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.

sys and os Modules

- The `sys` module provides a number of functions and variables that can be used to manipulate different parts of the Python runtime environment.
- The `os` module in Python provides a way of using operating system dependent functionality. The functions that the OS module provides allows you to interface with the underlying operating system that Python is running on – be that Windows, Mac or Linux.

Lab 17 - 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` or `help` to list the documentation for the following, and then create some data on which to test them. Start with `os_sys.py` in your data:
 - `os.getcwd`
 - `os.path.dirname` and `os.path.basename`
 - `os.path.exists`
 - `os.path.isdir` and `os.path.isfile`
 - `sys.exit` and `sys.argv`
- `sys.path` is a list. 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>

Importing

- You can create your own modules to import
 - Makes the code available to others
 - Unclutters your display
- Set the search path for imports
 - Permanently in system settings
 - Temporarily in the shell
 - Temporarily in the Python program

```
import sys
sys.path.append('/home/student/imports')
print(sys.path)
```

Lab 18

Using the program you created for Lab 16, separate the code for the classes from the main program. Place the code for the classes in a new directory. Give this code module a short name as directed in PEP 8. Import the code for the classes from that directory. Keep in mind whether you want to access the classes through dot notation or not. That will govern how you do the import.

Use the code on the previous slide to add the new directory to your path temporarily. Also, add a statement to your main program to import your class code. Get your program working paying special attention to the naming requirements. Also, examine the code in your new directory. Note the module ending with .pyc. We will discuss this when you are done.

Collections / Counter

- The collections module has a number of useful classes.
- We will use one of them as an example: Counter
- Counter is a dictionary subclass that has many of the attributes of a dictionary with a number of useful enhancements.
- As the name implies, it is built to aid counting.
- As with any class, it has to be instantiated.
- Normal dictionary rules/methods apply except:
 - fromkeys does not work
 - most_common(n) produces the n highest occurring keys
 - Incrementing nonexistent keys defines and initializes the key.
 - There are more details [here](#).

Unique Aspects of Counter

`x = Counter('himalayas')` # Here x becomes:

`Counter({'a': 3, 'i': 1, 'h': 1, 'm': 1, 'l': 1, 's': 1, 'y': 1})`

`y = Counter(list(x))` y contains all keys in x with a count of 1.

`x.subtract(y)` reduces all counts in x by 1 and retains all results

`x = x - y` reduces all counts by 1 and removes all items less than or equal to zero

`x.subtract(x)` reduces all counts to zero and retains all results.

`x = Counter()` creates an empty Counter or clears an existing one

- See mCounter.jpg in Samples

Lab 19

- a) Use the file `alice_in_wonderland.dat` to create a Counter. First, read the entire document into memory, and remove all whitespace using `translate/maketrans`. Then, use a Counter to determine the 20 most used characters in this document. Print out each character and its number of occurrences in descending order by use count. Be sure to make all cases the same.
- b) Alternatively, use `translate` and `split` to isolate each word in the book. Then use Counter and other tools such as sets to count the total words in the book, the unique words in the book, the number of words in the book that are not in `words.txt` and the top 20 words in the book by use counts. (See answers on the next slide)

Lab 19 Part B Results

```
Total words in book - 26695
Unique words in book - 2618
129 Book words not in dictionary
the.....1,644
and.....872
to.....729
a.....632
she.....541
it.....530
of.....514
said.....462
i.....410
alice.....386
in.....369
you.....365
was.....357
that.....280
as.....263
her.....248
at.....212
on.....193
all.....182
with.....181
```

assert Statement

- The assert statement is used as a debugging tool.
- At key places in your program, you test a condition.
 - If that condition fails, an AssertionError is raised.
 - Example: `assert x - y > 10`

```
Traceback (most recent call last):  
  File "<pyshell#2>", line 1, in <module>  
    assert x - y > 10  
AssertionError
```

- You can bypass assert statements
 - `python -O progname` – bypasses asserts

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)` execute 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
- `cl(ear)` 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 `pdb pdb2.py`
(in Windows – `python -m pdb 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__` module 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

Converting to Python Version 3

- Using the 2to3 module – use 2to3 as the command as with pdb
 - Ex: 2to3 /home/student/[progname.py]
 - Processes one program or an entire directory
 - Windows must use:
`python c:\python27\tools\scripts\2to3.py file_dir`
- Pertinent flags – default produces a report of changes
 - -w - create output files with fixes in the same directory.
Rename original files and keep as backup.
 - -n - suppress the creation of the backups mentioned above
 - -o - send output to a different directory. Use -n here.
 - -W - write all files even if there are no changes. Implies -w
Used mostly with -o to ensure entire tree is copied.

Converting to Python Version 3

- A Few Examples of Fixers:
 - **xrange** - Renames xrange() to range() and wraps existing range() calls with list where appropriate.
 - **raw_input** - Converts raw_input() to input().
 - **print** - Converts the **print** statement to the **print()** function.
 - **dict** - Fixes dictionary iteration methods. **dict.iteritems()** is converted to **dict.items()**, **dict.iterkeys()** to **dict.keys()**, and **dict.itervalues()** to **dict.values()**.
 - Find a complete list [here](#)
- Examples of 2to3:
 - 2to3 -w /home/student/progname.py
 - 2to3 /home/student/pyprogs -W -n -o /home/student/newdir

Lab 20

Your LabsData directory contains a program named `convert.py`

Use `2to3` to update only this program. Allow the backup version to be created. Observe the result. What changed?

Python IV

- Review
- [JSON](#) including [Pickle](#) and other alternatives
- [REST API](#)
- [Argparse](#)
- [Decorators](#)
- Subprocesses
- Threads
- Multiprocessing
- Test-driven development
- [Python Web Server Gateway Interface](#) (WSGI)

The End