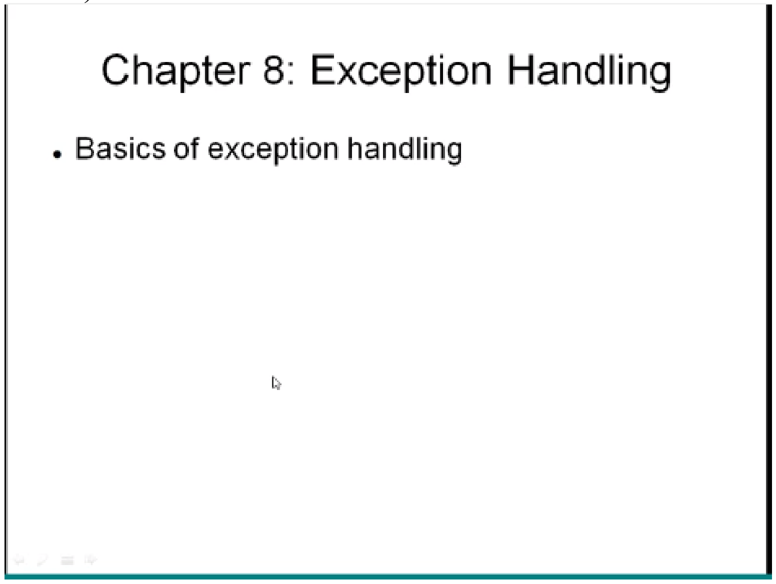
Hi everyone welcome to the LPS session , in this lecture we are going to continue to look at Python, just as a recap ,we covered the many things with Python ,as you may recall ,we started with the basic data structures ,we went through some of the control structures, we started talking about some of the advanced data structures like those lists and then , we went into the dictionary essentially , which is the associated ,associative arrays.

Then in the last lecture, we covered new items like λ , it is like a function generator and then we went over ,some of the default functions that are provided with as a part of Python, so I hope like coming, all those things are clear ,today we will , start with the actually ,another section, basically one more, this is I am going to talk about the exception handling.

(Refer Slide Time: 01:57)

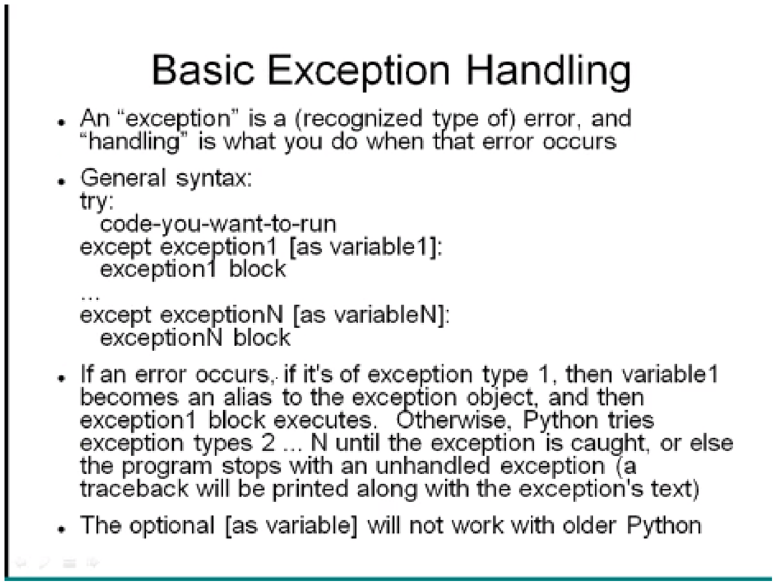


Chapter 8: Exception Handling

- Basics of exception handling

So this is like, when you have some exceptions and waited, inside code or how does the program handle it, how do we handle it, think like that ,that is what we talked about .

(Refer Slide Time: 02:07)



Basic Exception Handling

- An "exception" is a (recognized type of) error, and "handling" is what you do when that error occurs
- General syntax:

```
try:
  code-you-want-to-run
except exception1 [as variable1]:
  exception1 block
...
except exceptionN [as variableN]:
  exceptionN block
```

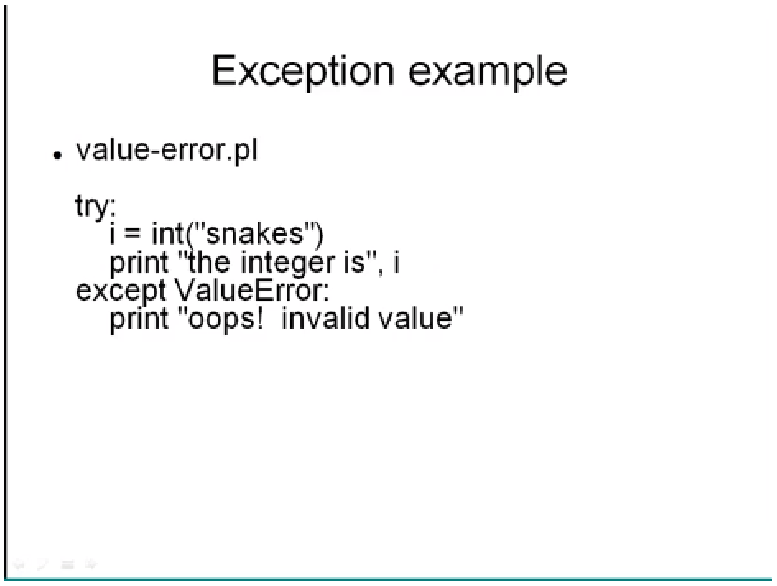
- If an error occurs, if it's of exception type 1, then variable1 becomes an alias to the exception object, and then exception1 block executes. Otherwise, Python tries exception types 2 ... N until the exception is caught, or else the program stops with an unhandled exception (a traceback will be printed along with the exception's text)
- The optional [as variable] will not work with older Python

So the exception in general is recognized ,type of error and handling is what you do, when that error happens , so the general syntax of exception handling is this, call called try, very similar to a function, but this is not a function, it is a predefined function and then basically ,what we say is ,we say try the code ,you want to run ,except order the exception ,so exception 1 and then it is exception 1, so for that what is the block of the code.

Then we do exception 2 and then we would do and all the way up ,to the last exception n and then we specify that as variable 1, so essentially like and this is the general syntax of the exception handling, so now, what this means is this code, means is that is an error , occurs and if it is of exception type 1 ,which is here , then we set the variable 1, it becomes an alias to the exception object .

So that is specified here and then this block of code ,starts executing, exception 1 block executes ,otherwise Python tries the exception types 2, all the way up to n ,until the exception is caught, or else the program stops ,with an unhandled exception ,the hash variable is an optional one and this will not work, with the order form ,this is like 3 and above .

(Refer Slide Time: 04:12)



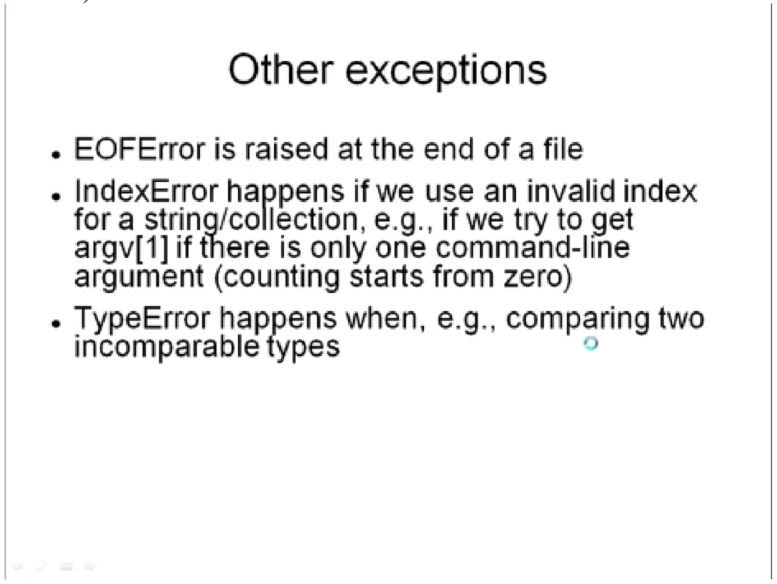
Exception example

- value-error.pl

```
try:
    i = int("snakes")
    print "the integer is", i
except ValueError:
    print "oops!  invalid value"
```

So let us look at a quick example, so value error ,this should be py, so here we say, try i =int (snakes) and then print the integer is something, except value errors ,here we omitted that as variable , print oops invalid value.

(Refer Slide Time: 05:11)

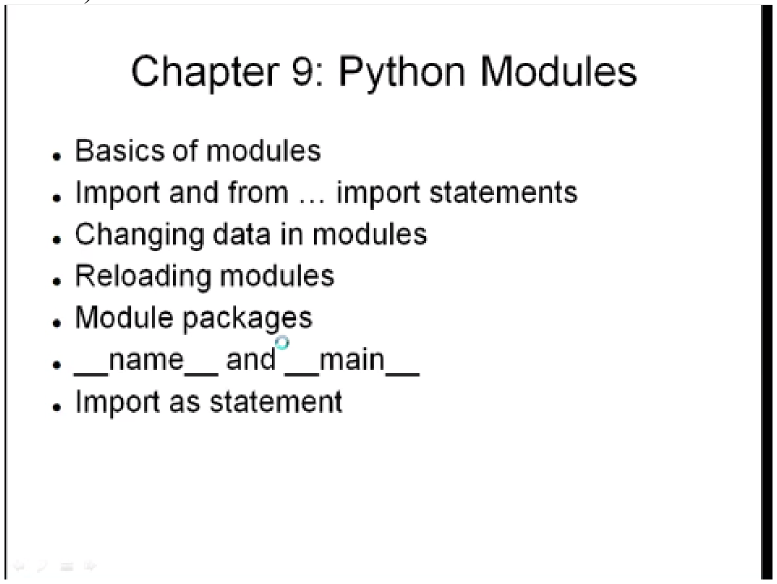


Other exceptions

- EOFError is raised at the end of a file
- IndexError happens if we use an invalid index for a string/collection, e.g., if we try to get argv[1] if there is only one command-line argument (counting starts from zero)
- TypeError happens when, e.g., comparing two incomparable types

Now ,there can be other exceptions ,one of the exception ,is this end of file exception , end of file error, is raised at the end of the file, the index error ,happens if we use an invalid index or a string or a collection ,example if we try to get arg v1, there is only one command line argument, which is basically like arg v 0 ,then it generates an index error, another one is called type error, which is if you are trying to compare two different types ,incomparable types like string versus float or something ,again that is the type error.

So essentially like I mean, in general like a value error ,end of file error and then an index error and then the type error, so these are the general exceptions ,so that you want to keep in mind .
(Refer Slide Time: 07:06)



Chapter 9: Python Modules

- Basics of modules
- Import and from … import statements
- Changing data in modules
- Reloading modules
- Module packages
- __name__ and __main__
- Import as statement

Now let us look at ,some of the Python modules ,so in this section ,we are going to talk about, the basic of modules , import and from ,import statements and then how do we change data inside the module, we will also talk about reloading the modules, then we will talk about the module packages, then name and the main, then finally how do we import as statements, so now let us look at the Python modules .

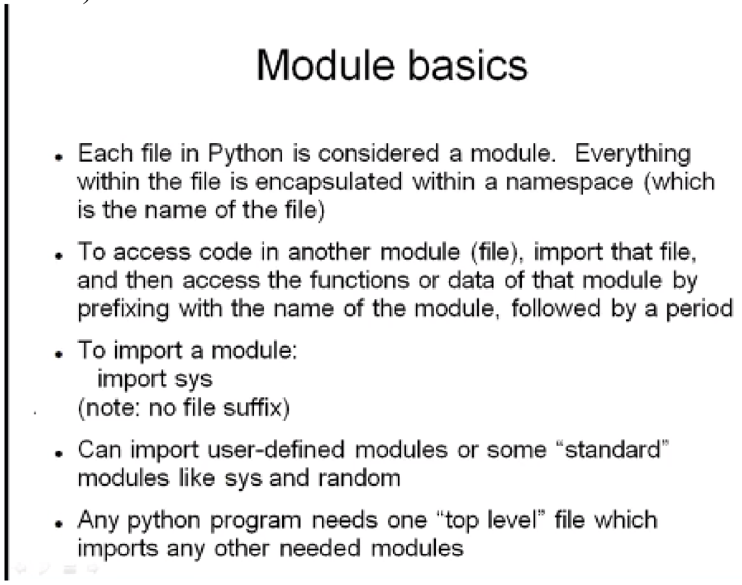
So a couple of more things ,regarding ,before we go into the modules, I want to also highlight some of the other things, about exceptions ,so we talked about some of the things basically, so the other ones ,essentially you can also have standard error, which is basically, which can be like in any ,all the built-in exceptions except, the stop iteration generator ,exit keyboard interrupt ,things like then our system exit.

They are all the standard the error and then we all can also get an arithmetic error, which is the essentially like I mean ,various types of arithmetic operations, which can be generated from arithmetic operations like overflow error ,0 division error or floating point level and then there is also buffer error ,which is essentially like I mean the buffer related operations cannot be performed, you can get a buffer error.

And then there is also look up error ,which is the index error ,is one of them ,since it or even the key error is also the other one, is only classified, under look up error and then the things like IO error ,even this end of file error, these are all under the environment error, which is concerning with outside Python system, so IO error ,OS error and also be with them environmentally and there are some assertion errors, basically when an assert statement, attribute errors then an attribute reference or attribute assignment page.

We also have talked about the floating point error, basically when the floating point operation fails and then there is the generator exit, so all these things comes under the exception class of the Python ,so they are basically like, it is a base class, that is for all the exceptions , so now let us look at the modules ,Python modules .

(Refer Slide Time: 11:32)



Module basics

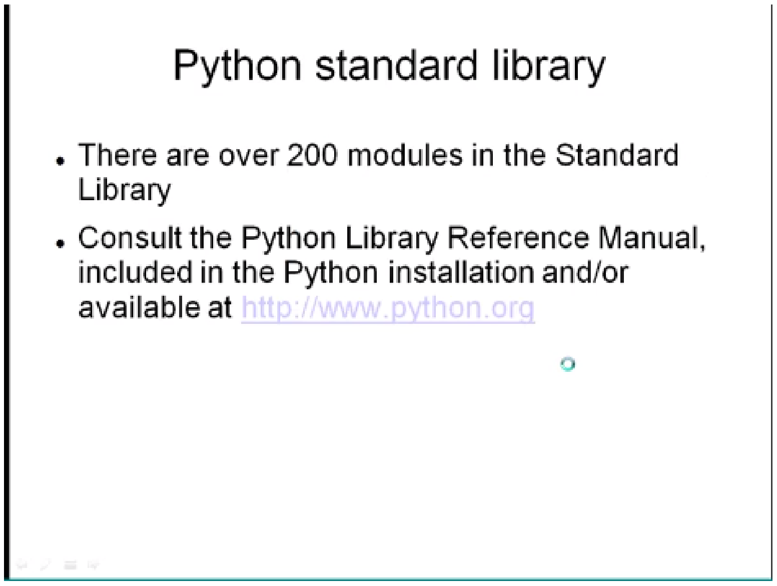
- Each file in Python is considered a module. Everything within the file is encapsulated within a namespace (which is the name of the file)
- To access code in another module (file), import that file, and then access the functions or data of that module by prefixing with the name of the module, followed by a period
- To import a module:
  import sys
  (note: no file suffix)
- Can import user-defined modules or some "standard" modules like sys and random
- Any python program needs one "top level" file which imports any other needed modules

So what is the module first of all ,each file in a in Python is considered a module ,so this module actually like it , we say it in Perl as well, but we did not see in Tcl ,we did not see that much, but in Perl ,we saw the modules essentially, so everything within the file is encapsulated within the namespace ,which is basically the name of the file okay ,so to access code in another module ,another file , we need to import that file and then access the functions of the data of that module .

We can do it, by prefixing the name of the module, followed by a period, but to import a module, we just use the function called input and with the argument as sys ,so one thing you will notice basically ,if there is no suffix for this particular module, we can also input user-defined module or some standard modules like sys and random .

So in this case like this is like a standard module, but you can also like to import user defined modules, any Python program needs one top-level file, which imports any other needed modules, so the way it is organized as top-level file ,which inputs other files.

(Refer Slide Time: 13:18)

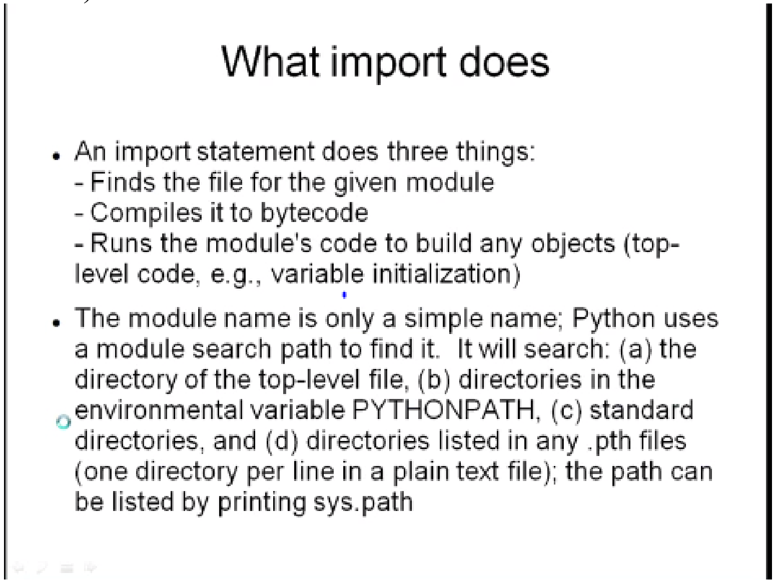


Python standard library

- There are over 200 modules in the Standard Library
- Consult the Python Library Reference Manual, included in the Python installation and/or available at http://www.python.org

So in the Python standard library, there are over 200 modules and so essentially, like I mean you can go to python.org and then you can, consider the consult the Python library reference manual and for all these different modules in the standard library .

(Refer Slide Time: 14:02)



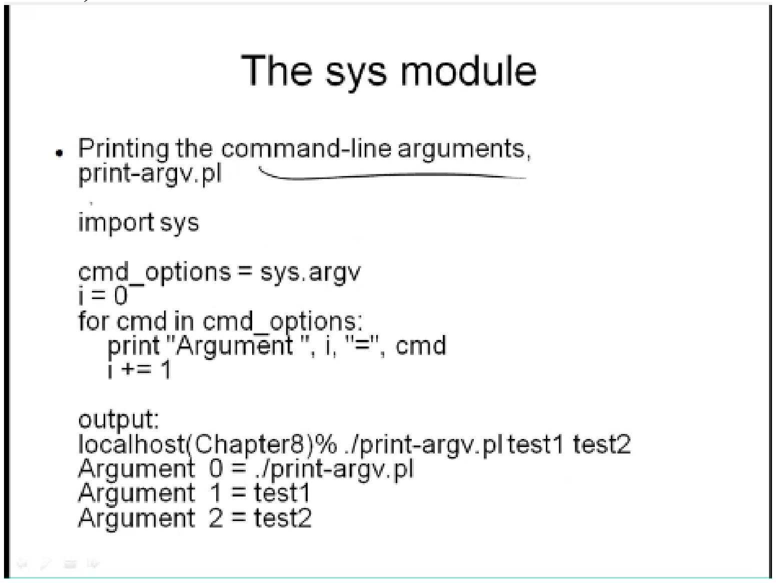
What import does

- An import statement does three things:
  - Finds the file for the given module
  - Compiles it to bytecode
  - Runs the module's code to build any objects (top-level code, e.g., variable initialization)
- The module name is only a simple name; Python uses a module search path to find it. It will search: (a) the directory of the top-level file, (b) directories in the environmental variable PYTHONPATH, (c) standard directories, and (d) directories listed in any .pth files (one directory per line in a plain text file); the path can be listed by printing sys.path

So now ,what does the import do essentially ,so the import statement does, three things, it finds the file for the given module, it compiles into a byte code and runs the module code to build any objects, basically the top-level code and then variable initialization, the model name is only a simple name, Python also uses a module search ,path to find it ,it will search the directory of the top level module or the top-level file ,directories inside the environmental variable, called the Python path .

And then the standard directories and then finally the directories lists, listed in any of the path files, that is in your directory and then, here it is basically like one directory per line in a plain text file , essentially that is how, one specifies path file , the path can be listed ,by printing sys. path, so if you say sys.path, it will list the path.

(Refer Slide Time: 15:28)



The sys module

- Printing the command-line arguments, print-argv.pl

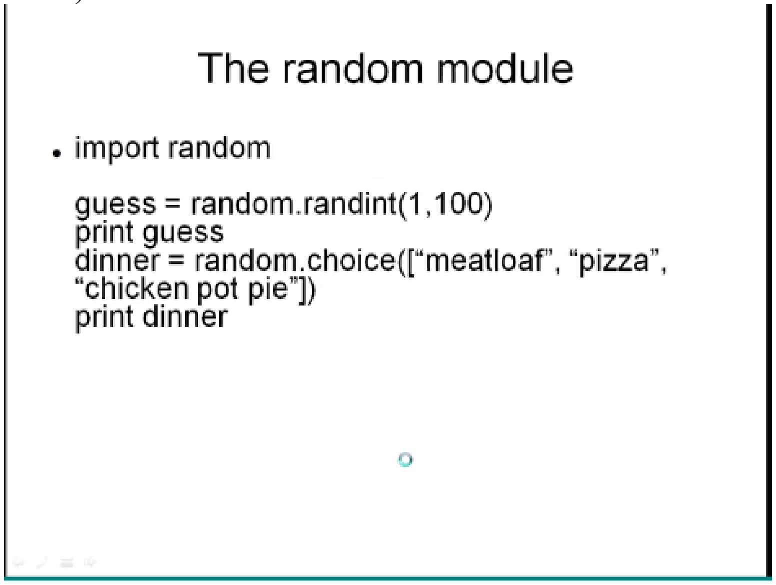
```
import sys

cmd_options = sys.argv
i = 0
for cmd in cmd_options:
    print "Argument ", i, "=", cmd
    i += 1

output:
localhost(Chapter8)% ./print-argv.pl test1 test2
Argument  0 = ./print-argv.pl
Argument  1 = test1
Argument  2 = test2
```

So here is an example, for printing the command-line arguments and argv.pl ,so here we import the sys and then the command options, are not arg v and then basically like for the initialize the variable I and then for command in ,command options essentially which is all the forms, we just print the argument I and then the command and it basically prints ,once we specify this and then print arg v.pl and then test 1 and test2 .

(Refer Slide Time:16:18)

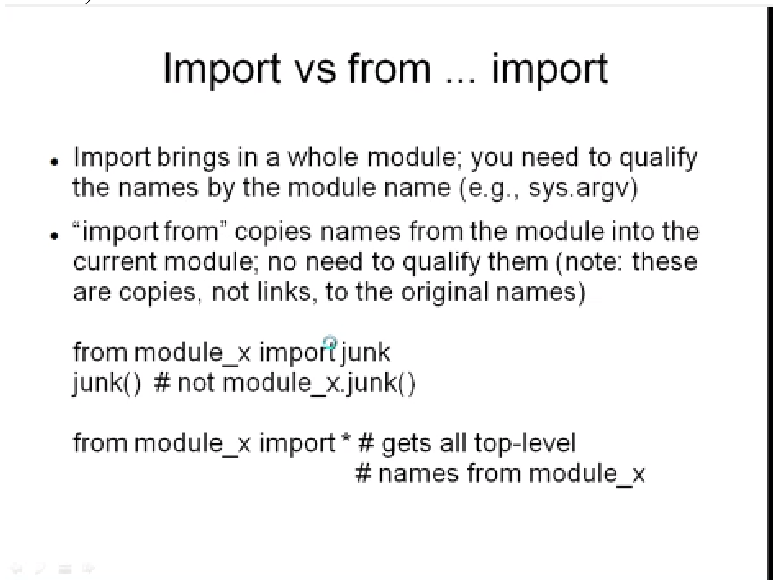


The random module

- import random

```
guess = random.randint(1,100)
print guess
dinner = random.choice([“meatloaf”, “pizza”,
“chicken pot pie”])
print dinner
```

Now there is another module called random ,so random also can be imported ,so we go to random and then we say basically random. rand int between 1 and 100, so this is essentially if it is a random number generator between 1 & 0, I mean sorry 1 and100 and then we can print that and then you can also like specify ,a random got choice and then you, can pass a list and then we can make it to print, the dinner and then it randomly picks one .
So here we notice basically, like this random function and we also say like rand int, to make sure that this output of this ,one and here we just specify choice and then basically like some strings and then essentially like gets printed on.
(Refer Slide Time: 17:31)



Import vs from ... import

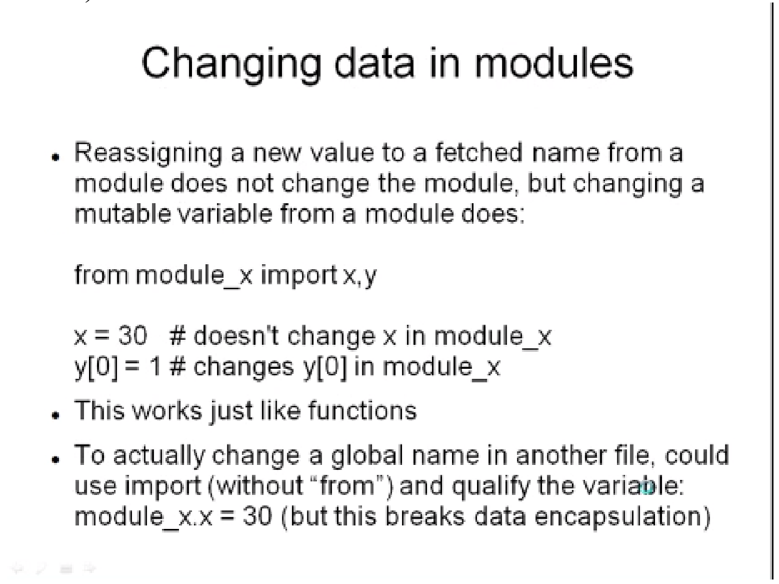
- Import brings in a whole module; you need to qualify the names by the module name (e.g., sys.argv)
- "import from" copies names from the module into the current module; no need to qualify them (note: these are copies, not links, to the original names)

```
from module_x import junk
junk()  # not module_x.junk()

from module_x import * # gets all top-level
                       # names from module_x
```

So now let us look at , , the difference between import and from , from import basically, so from this whole thing ,so when we say import, import brings in a whole new module essentially the whole module ,essentially and we need to qualify the names by modeling, so that is using this sys.arg v , but import from copies the names, from the module into the current module, so no need to qualify that ,we do not need to actually qualify with this additional names.
So it is actually brought into the current model itself so and notice that actually these are fully copies basically not links so essentially you can change and basically like it is not going to overwrite anything changes to the original ones, so you have a your own copy and then your copy will be different from the one inside the module itself so here an example, from module X the input junk and then we directly call junk we do not call as Model X or junk anymore.
So now when we specify this command the junk actually now gets transferred and then this way it is copied and kept it as part of your module and not part of the X and then you can say

basically later from module X import star that gets all the top-level mod names from modeling X.
(Refer Slide Time: 20:07)



Changing data in modules

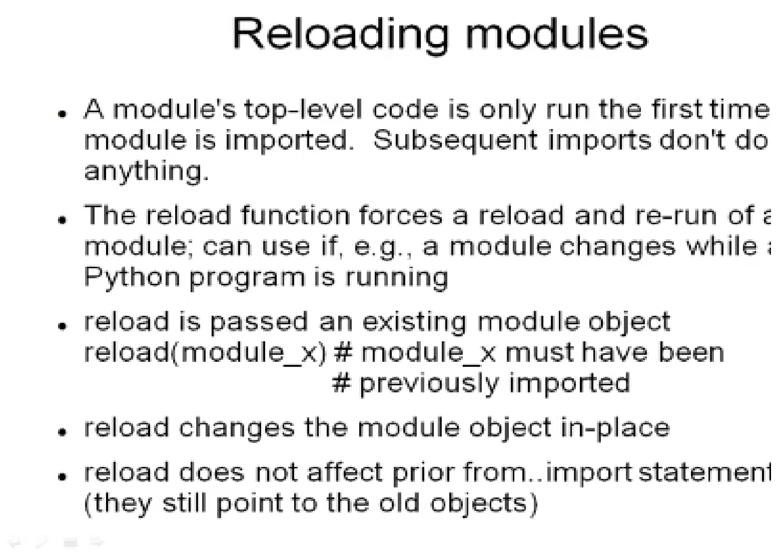
- Reassigning a new value to a fetched name from a module does not change the module, but changing a mutable variable from a module does:

```
from module_x import x,y

x = 30   # doesn't change x in module_x
y[0] = 1 # changes y[0] in module_x
```

- This works just like functions
- To actually change a global name in another file, could use import (without "from") and qualify the variable: module_x.x = 30 (but this breaks data encapsulation)

Now the change, changing the data inside model for something so you want to reassign a new value to fix name from module from a module which does not change the module with changing the mutable variables from the module, so essentially like I mean a module X import X Y then you say x = 30 it does not change X in the module x so this is something that we talked about earlier but if you put Y 0 =1 that actually changes Y0 in Model X, because now it has no bearing of this zero inside this particular you are your own module and it needs to go back to the Model X .
To find out what is y 0 and then change that so this is similar to like module exactly like the functions so to actually change a global name in another file we could use import without the from and qualified available, but again this breaks the data encapsulation because now we say like Model X . X 30 and then any kind of object encapsulation that the module provides is compelling.
(Refer Slide Time: 21:43)

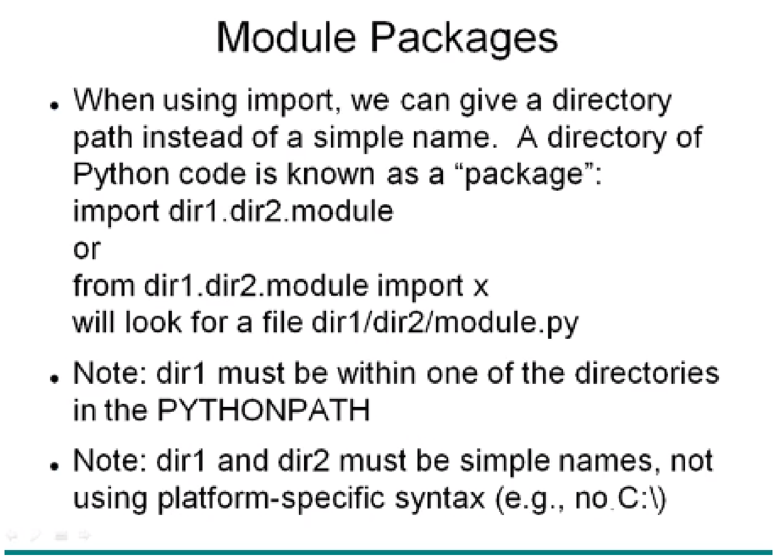

Reloading modules

- A module's top-level code is only run the first time module is imported. Subsequent imports don't do anything.
- The reload function forces a reload and re-run of a module; can use if, e.g., a module changes while a Python program is running
- reload is passed an existing module object
```
reload(module_x) # module_x must have been
                 # previously imported
```
- reload changes the module object in-place
- reload does not affect prior from..import statement (they still point to the old objects)

Now how do we reload modules the model produced top-level code is only run the first time when the model gets imported subsequent imports do nothing basically, so the reload function force the cel-3 load and rerun of that module ,so you can use if there is a module changes while the Python program is run so essentially like I mean if you change the variable and basically you have to be import the module then at the point you can use very low and reload is passed an existing model object for example reload Model X and then the Model X must be must have been previously imported.

So it would not you cannot do an import with a reload comment and then the reload changes module object in place basically like whatever that was imported it changes that essentially and it does not affect prior from import statement, so once they are copied into your own particular module those will not change even with real only the import the regular important they did only one that will change and they still point to the old objects. If you do another from import of course that will change basic app.

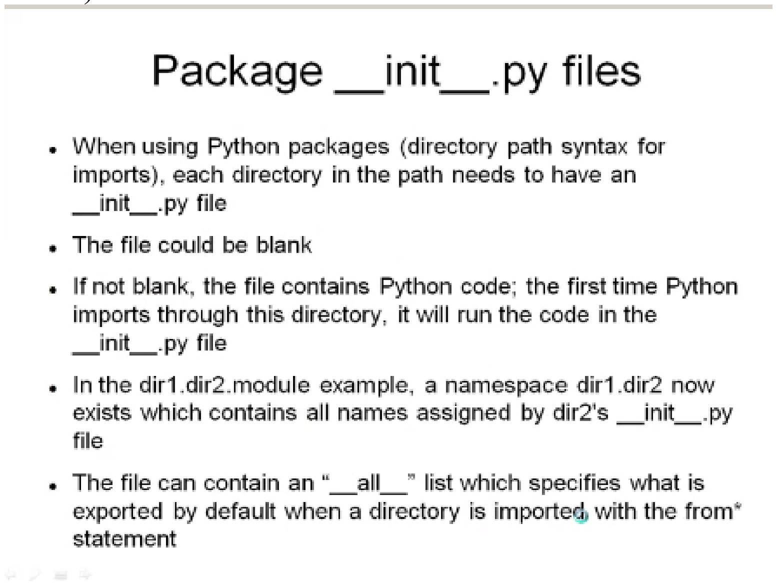
(Refer Slide Time: 23:18)



Module Packages

- When using import, we can give a directory path instead of a simple name. A directory of Python code is known as a "package":
  import dir1.dir2.module
  or
  from dir1.dir2.module import x
  will look for a file dir1/dir2/module.py
- Note: dir1 must be within one of the directories in the PYTHONPATH
- Note: dir1 and dir2 must be simple names, not using platform-specific syntax (e.g., no C:\)

So now let us talk about the module packages so when using the input we can give the directory path instead of a simple , so the directory of Python code is known as the package so for example here input dir1,dir 2 module so are we can also say like from exactly 1 dot directed to module input X so essentially what that means is basically look for the file / 31 / directory to and then modify and one thing is directory one must be within one of the directories in the Python .
So the Python path is now important similar to what fine regular import models and the other thing is also like that we won and directly to they should be simple names not the platform specific syntax there is P: /.
(Refer Slide Time: 24:29)

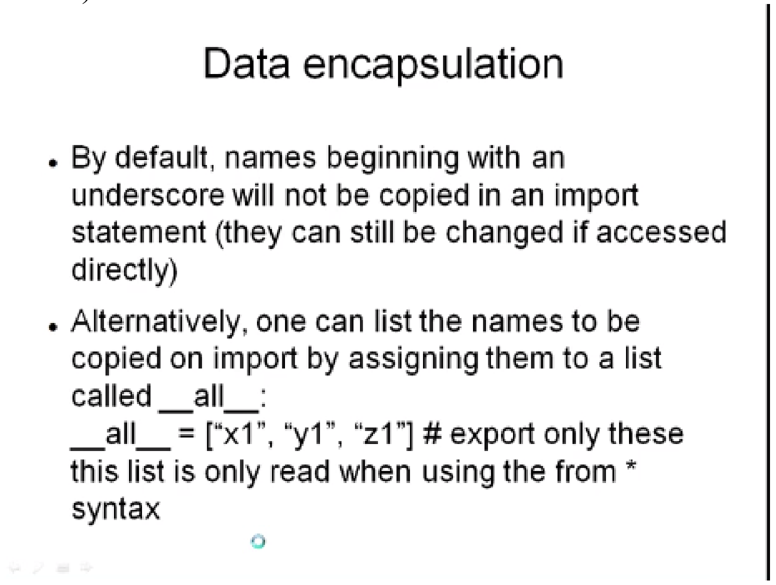


Package __init__.py files

- When using Python packages (directory path syntax for imports), each directory in the path needs to have an __init__.py file
- The file could be blank
- If not blank, the file contains Python code; the first time Python imports through this directory, it will run the code in the __init__.py file
- In the dir1.dir2.module example, a namespace dir1.dir2 now exists which contains all names assigned by dir2's __init__.py file
- The file can contain an "__all__" list which specifies what is exported by default when a directory is imported with the from* statement

Now underscore in it underscore, underscore. PY these are some of the package files so when we use the Python packages basically the directory that syntax for the imports each directory in the

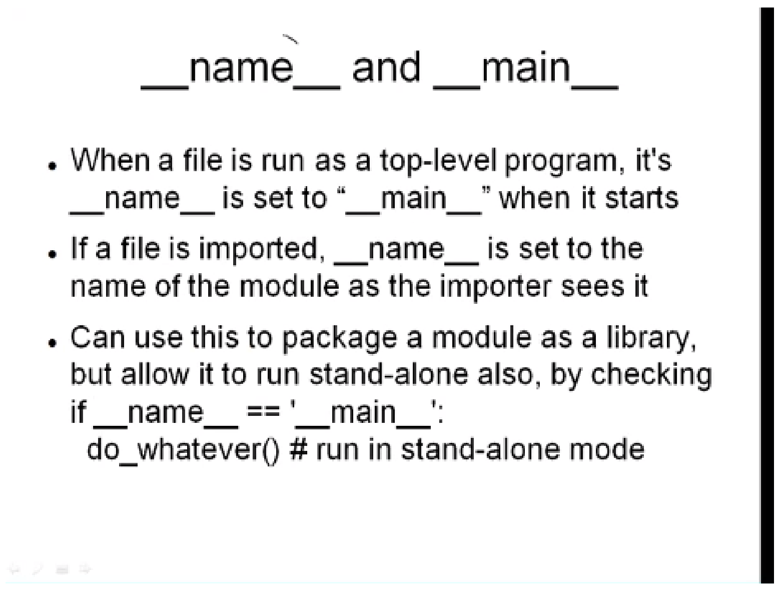
back needs to have this underscore, underscore in it under former school not PY file this files could be this black basically I mean if it is it is nothing basically or if it is not blank the file contains the Python code the first time python imports through the directory it will run the code in the underscore ,underscore in a thunderstorm of the score but Python.
So in the directory one directly to model example the namespace dir 1 ,dir 2 now exists which contains all the names assigned by the Directorate to underscore ,underscore in it understands the Python the file can contain and underscore, underscore all underscore ,underscore list which specifies what it exported by default when the directory is imported bit the form statement.
(Refer Slide Time: 26:01)



Data encapsulation

- By default, names beginning with an underscore will not be copied in an import statement (they can still be changed if accessed directly)
- Alternatively, one can list the names to be copied on import by assigning them to a list called __all__:
  __all__ = [“x1”, “y1”, “z1”] # export only these
  this list is only read when using the from * syntax

So now this actually gives us another concept of data encapsulation so by default names beginning with an underscore will not be copied in an import statement they can still be changed if access directly so that is no issues alternatively one can still want a list the names to be pipe on import by assigning them to a list called underscore, underscore all underscore number four so for example here we decided on the phone will call on underscore school and this list x1 y1 + V 1 and 2 mean that basically import only these this list is read-only only like we need to put this these basic x 1 y 1 0.
And this list is the read-only and when it is using the from star syntax that is the one becoming both import from syntax that we saw so this is this list is read-only then you use this from stuff so that when you use the import from and then start it only imports this even though if they have variable.
(Refer Slide Time: 27:32)

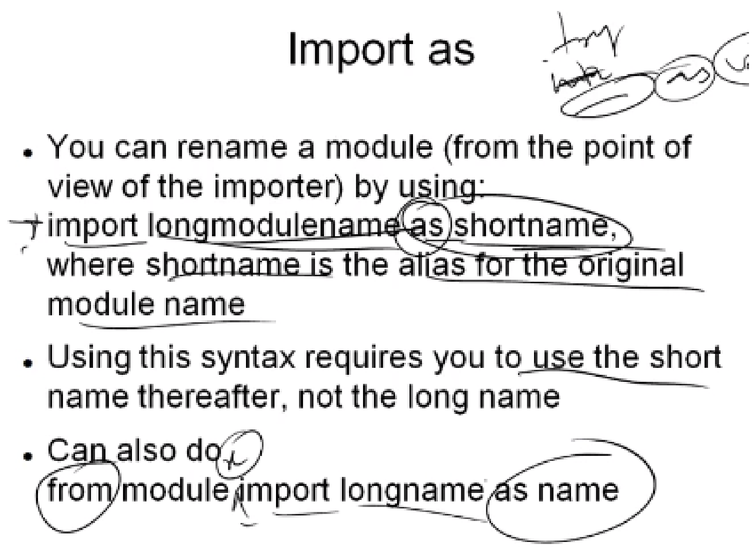


__name__ and __main__

- When a file is run as a top-level program, it's __name__ is set to "__main__" when it starts
- If a file is imported, __name__ is set to the name of the module as the importer sees it
- Can use this to package a module as a library, but allow it to run stand-alone also, by checking

```
if __name__ == '__main__':
  do_whatever() # run in stand-alone mode
```

So now the same thing basically like on the four name on the score and underscored me in on this forum so whenever file is run as a top-load program its underscore name underscore is set to underscore main underscore then it starts , so this is how so this is the concept which is very similar in C language in C always the execution starts with this function called name and it only execute that and then any other functions are underneath that similarly in Python basically the underscore and underscore name.
And the score is set to main when it starts so that it runs that as the starting point but if the file is important the underscore name underscore is set to the name of the module as the importer says it and can use this to package a module as a library but allow it to run as a standalone off by checking if the name the underscore name on the score is the same as underscore name on the scope and do whatever basically that will run in the standalone .
(Refer Slide Time: 29:00)



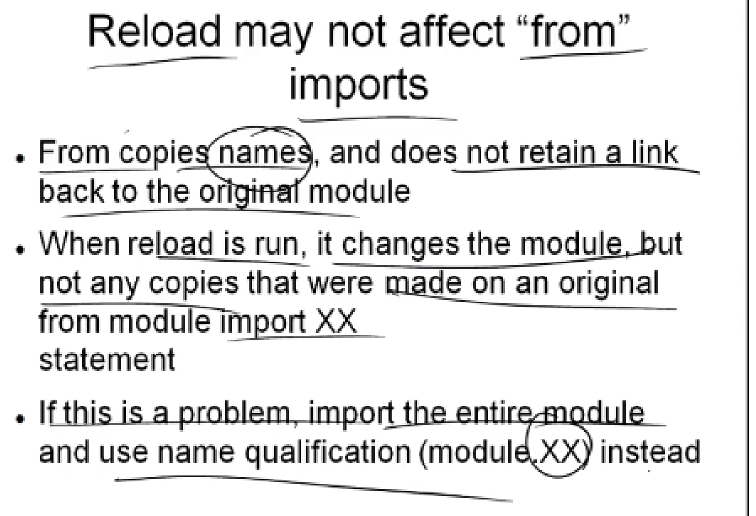
Import as

- You can rename a module (from the point of view of the importer) by using:
  import longmodulename as shortname, where shortname is the alias for the original module name
- Using this syntax requires you to use the short name thereafter, not the long name
- Can also do
  from module import longname as name

So now the next one is this input ass we can rename a module by using import long model name as short model that the short name is just an alias for the wording model we saw this thing in the very first slide when we with the example for importing we said the import or actually like in the exception handling so try and then we say basically exception as variable one so there that long name is assigned to the shortening essentially visiting the alias so the same thing is applicable even to the import statement so import non-model name as shortening and then the short name is the alias forward.

So when we start using this interacts with it mean like and we do not have to specific moment you can also use the same as syntax or the from this is so from module import long name as the shot that is another thing that we can do yeah basically here X module .

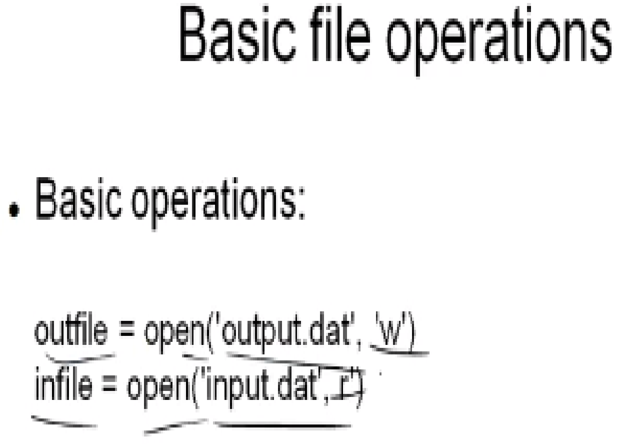
(Refer Slide Time: 30:28)



Reload may not affect "from" imports

- From copies names, and does not retain a link back to the original module
- When reload is run, it changes the module, but not any copies that were made on an original from module import XX statement
- If this is a problem, import the entire module and use name qualification (module.XX) instead

And we saw this thing basically would not go below the reload may not affect from imports Ascension so when we do from copy so basically names the and does not retain the link back those involved so that is why we do a reload you would not get the new one for those modules that are imported as front so when the reload is run it changes the module but not any copies that were made on the original from the module input from modeling products statement if this is this becomes a problem.
Then you need to import my entire module and then just use the name qualification just module. XXX xxxx form this that kind of name instead of this from statement.
(Refer Slide Time: 31:44)

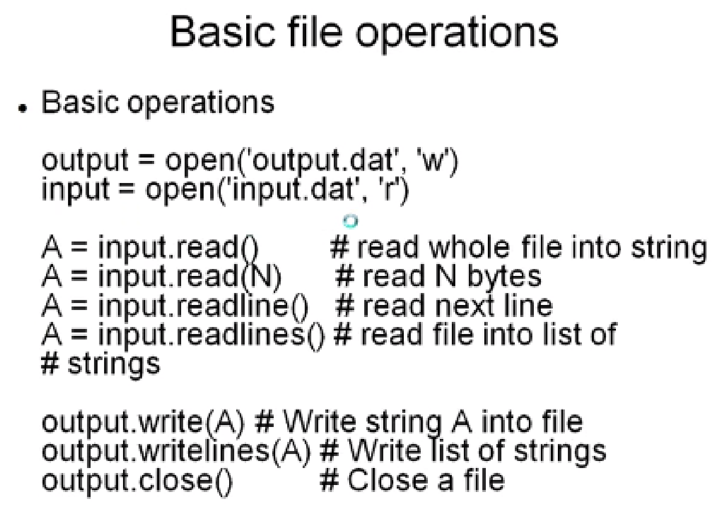


Basic file operations

- Basic operations:

```
outfile = open('output.dat', 'w')
infile = open('input.dat','r')
```

So now we come to the final phone section here we are going to just talk about some basic sign operations you so quick commands how do we open a file the command is open and then we specify the filename over more and whether the file name is just the Python string the mode is also a string basically and then it is string are for reading the W for writing and a for a file so the basic operation basically out file open out dad with right option in file open input dot there and then with the read option simple stuff we saw this kind of thing both in TCL and Perl quality.
(Refer Slide Time: 32:41)



Basic file operations

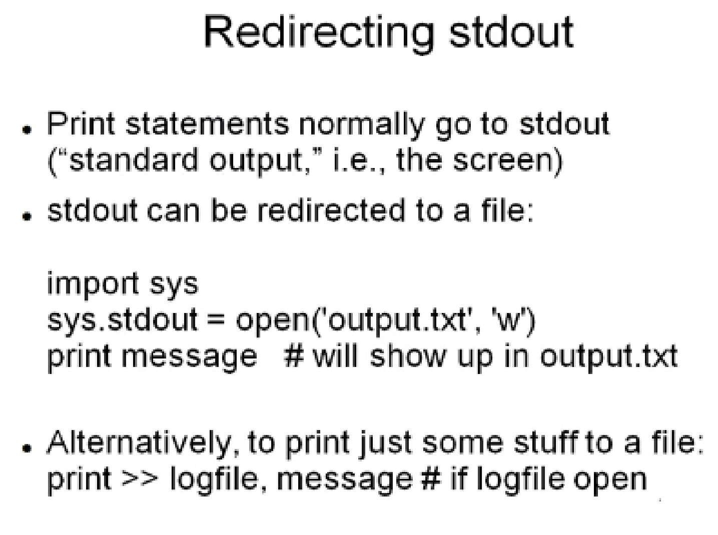
- Basic operations

```
output = open('output.dat', 'w')
input = open('input.dat', 'r')

A = input.read()       # read whole file into string
A = input.read(N)      # read N bytes
A = input.readline()   # read next line
A = input.readlines() # read file into list of
# strings

output.write(A) # Write string A into file
output.writelines(A) # Write list of strings
output.close()         # Close a file
```

And then some of the basic operations so once we open the file we specify input read and then that reads the whole file into a strength this also we saw like in the previous examples and then we can also read by number of bytes essentially like so that is again read N the N specifies the number of bytes so it reads ,so many bytes from the file and get sit and read line is to read the next line and we can also specify read lines to read the files into a list of strings and then similarly for writing some output basically use the output dot right and then we specify the string a and then the string a is written into the form.
And then we can say like right lines A to write a list of strings into the E and then we use the clothes function or close closer file.
(Refer Slide Time: 33:36)



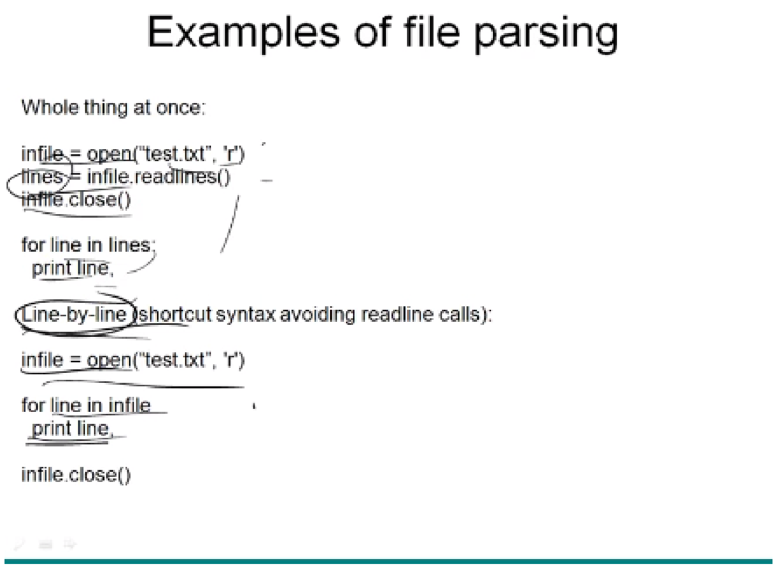
Redirecting stdout

- Print statements normally go to stdout ("standard output," i.e., the screen)
- stdout can be redirected to a file:

```
import sys
sys.stdout = open('output.txt', 'w')
print message   # will show up in output.txt
```

- Alternatively, to print just some stuff to a file:
  `print >> logfile, message # if logfile open`

Now how do we redirect the standard out so the print statement normally go to the Standard out mount the standard output is basically the screen and then we can actually redirect the STD out into a file so here we basically the input versus module and then we specify standard out is this particular file called output or text and they open it with the right option now when we specify the printer message it will not show up in your screen.
But it directly goes into this output on text alternatively we can also print something basically to a file using the print double arrow and then we specify the file and then the movies and then for this the only condition blockers.
(Refer Slide Time: 34:58)



Examples of file parsing

Whole thing at once:

```
infile = open("test.txt", 'r')
lines = infile.readlines()
infile.close()

for line in lines:
  print line,
```

Line-by-line (shortcut syntax avoiding readline calls):

```
infile = open("test.txt", 'r')

for line in infile
  print line,

infile.close()
```

So here are some examples of file passing this by passing like. I mean we saw some Perl lot of times so here how do we do it in Python they specify the in file base again what is the in file with what kind of option and then we start reading the lines this is so the lines contain and list of the strengths of all go the lines in the 10 in this file and then we can get there after we read it we can disclose it now to print the line essentially like we use the for statement and then we print line that prints one line at the time.
And then there is also like a shortcut syntax to avoid the read line calls this is a line the line so we open the file and then we can play like the maybe so here once we specify the line by line essentially like in just read the lines or one by one. So here we can directly be with so in file and then print the slides.
So just to recap today we saw three major things basically the first one is on the exception handling how do we handle exceptions and we cover many different types of exception and in general the syntax try and catch essentially the try syntax or the exception handling then we also

cover the various modules module commands, the basics of module how to import modules how to use the from import comments then we also talked about the various types of modules like this is module axis module and random module.
And then we went through like how do we reload modules what are the effects basically and then all we depend packages and then the what are the same rules that govern the package specification and then how do we actually change what getting booted and what gets run and then we went through the file operations basically ,how do we open the file or we close the file how do we open it as read-only with us right.
And then how do we actually do operation system or we read from a file and how do we write out into a file and then also like how do we redirect some of the standard outs files and then we also went through the file parsing essentially so that is pretty much it for today we will pick up the stuff from this point in the next class thank you very much.