COMP47670

Next Steps in Python

Slides by Derek Greene

UCD School of Computer Science

Overview

- File Input/Output
- Error Handling
  - Python Error Messages
  - Exceptions
- Python Modules
  - Built-in modules
  - Basic Mathematical and Random Functions
  - Accessing Files and Directories
  - Writing Python Scripts
  - Command Line Arguments

File Input/Output

- Files are special types of variables in Python, which are created using the `open()` function. Remember to `close()` the file when you are finished!

```
f = open( "<filepath>", "<action>" )
      .....
f.close()
```

"r": read
"w": write
(default is read)

- **Reading files:** After opening a file to read, you can use several functions to access plain-text data:

`read()`	read the full file
`readline()`	read a full line from a file
`readlines()`	read all lines from a file into a list

Need to strip line endings

```
f = open("test.txt", "r")
lines = f.readlines()
f.close()
for line in lines:
    line = line.strip()
    print(line)
```

Read all lines from a file into a list

Example: Reading Files

- Read a list of names and student numbers, storing the information in a dictionary.

Input: students.txt

```
17211426,Stephanie Gale
16212133,Jill Doyle
13388136,Pat Gilbert
17211824,Daryl Bishop
16216364,Carlos  Alvarado
17211833,Alison Rogers
17212834,Neil Smith
13312141,Sandra Wright
```

```
register = {}
fin = open("students.txt","r")
lines = fin.readlines()
fin.close()
for line in lines:
   line = line.strip()
   parts = line.split(",")
   student_id = int(parts[0])
   fullname = parts[1]
   register[student_id] = fullname
```

Need to strip line endings

- Display the new contents of the dictionary:

```
for sid in register:
   print( "%d -> %s" % (sid, register[sid]) )
```

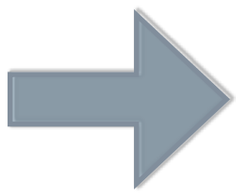
```
17211426 -> Stephanie Gale
16212133 -> Jill Doyle
13388136 -> Pat Gilbert
17211824 -> Daryl Bishop
16216364 -> Carlos Alvarado
17211833 -> Alison Rogers
17212834 -> Neil Smith
13312141 -> Sandra Wright
```

Working with Files

- **Writing files:** After opening a file to write, use the `write()` function to output strings to the file.

```
names = ["Mark","Lisa","Alice","Bob"]
f = open("out.txt","w")
for name in names:
   f.write( name )
   f.write( "\n" )
f.close()
```

Need to explicitly move to next line



out.txt

```
Mark
Lisa
Alice
Bob
```

- Note: By default Python will overwrite an existing file with the same name if it already exists.
- To add data to the end of an existing file, use append mode "a" when opening the file:

```
f = open("out.txt","a")
```

Indicates open in append mode

Example: Writing Files

- Read a list of lines from one file, write the contents back out to a second file with an additional prefix.

Open two files: one to read ("r"), one to write ("w")

```
fin = open("sample.txt","r")
fout = open("modified.txt","w")
for line in fin.readlines():
   fout.write("Copy: ")
   fout.write(line)
fin.close()
fout.close()
```

Note that the lines already end with a new line character

Input: sample.txt

```
County Dublin
County Galway
County Limerick
County Louth
County Wexford
```

Output: modified.txt

```
Copy: County Dublin
Copy: County Galway
Copy: County Limerick
Copy: County Louth
Copy: County Wexford
```

Writing Files + String Formatting

- We can write a variety of Python variables into a text file on multiple lines.
- Note we must convert values to strings before calling `write()`
- We can do this either using type conversion or using string formatting.

```
year = 2013
d = {"a":3.0, "b":4.5, "c":9.87}
fout = open("data.txt","w")
fout.write( str(year) + "\n" )
for key in d:
    fout.write(key + " " + str( d[key] ) + "\n")
fout.close()
```

```
year = 2013
d = {"a":3.0, "b":4.5, "c":9.87}
fout = open("data.txt","w")
fout.write( "%d\n" % year )
for key in d:
    fout.write("%s,%.1f\n" % (key, d[key]))
fout.close()
```

Output: data.txt

```
2013
a,3.0
b,4.5
c,9.9
```

Python Error Messages

- A key programming task is debugging when a program does not work correctly or as expected.
- If Python finds an error in your code, it raises an exception.
  e.g. We try to convert incompatible types
  e.g. We try to read a non-existent file
  Also... When we have invalid syntax in our code (a "typo")

```
number = int("UCD")
```

```
Traceback (most recent call last):
  File "test.py", line 1, in <module>
    number = int("UCD")
ValueError: invalid literal for int() with base 10: 'UCD'
```

Where the error occurred

Type of exception that has occurred

Text describing the error

Python Error Messages

```
d = {"Ireland": "Dublin"}
d["France"]
```

```
Traceback (most recent call last):
  File "test2.py", line 2, in <module>
    d["France"]
KeyError: 'France'
```

Where the error occurred

Type of exception that has occurred

```
def showuser(username):
  print(user_name)

showuser("bob")
```

```
Traceback (most recent call last):
  File "test3.py", line 4, in <module>
    showuser("bob")
  File "test3.py", line 2, in showuser
    print(user_name)
NameError: name 'user_name' is not defined
```

Error originated here

Exception Handling

- By default, an exception will terminate a script or notebook.
- We can handle errors in a structured way by "catching" exceptions. We plan in advance for errors that might occur...

General Format

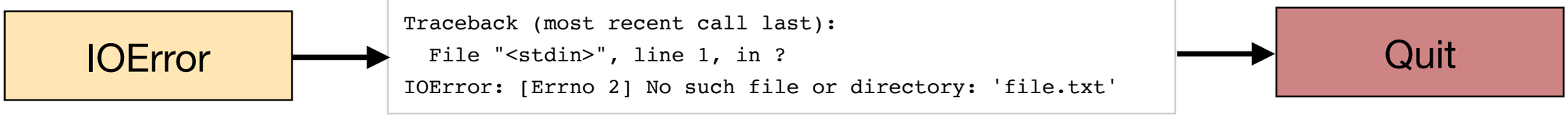
```
try:
   <block of code>
except <errorType>:
   <error handling block>
```

```
try:
    f = open("file.txt","r")
except IOError:
    print("Got error, but continuing anyway")
```

Code where error might occur

Error handling code

Without exception handling



With exception handling

More Complex Exception Handling

- We can get details about the specific cause of the exception - i.e. the error message.

```
try:
    x = int("ucd")
except ValueError as e:
    print("Error:",e)
```

```
Error: invalid literal for int()
with base 10: 'ucd'
```

- Try statements can include an optional `finally` clause that always executes regardless of whether an exception occurs.

```
try:
    x = int(some_string)
except ValueError:
    print("Conversion error")
finally:
    print("Always print this")
```

- Multiple except clauses: A try statement can check for several different exception types in sequence.

```
try:
    x = int(some_string)
    answer = x/y
except ValueError:
    print("Conversion error")
except ZeroDivisionError:
    print("Dividing by 0!")
```

Example: Exception Handling

- Common file input tasks required handling the case where we try to read from a file path that does not exist...

```
file_path = "/home/user/data.csv"
try:
    fin = open(file_path,"r")
    content = fin.read()
    print(content)
    fin.close()
except IOError:
    print("Unable to read from file", file_path)
finally:
    print("Process complete")
```

- If `file_path` does not exist, the except code block will be run:

```
Unable to read from file /home/user/data.csv
Process complete
```

Python Modules

- Module: A single file of Python code, often containing functions and variables related to a particular programming task.
- Accessing functions in a module requires first importing the module for use in the current Python environment. Two different ways to do this.

```
import <module_name>
```

Import the whole module in its entirety

```
from <module_name> import <something>
```

Import a subset of functionality i.e. just certain functions or variables which we require

- Examples of imports:

```
import math
import sys, os
```

Import whole modules. Note we can import multiple modules on each line.

```
from sys import exit
from math import sqrt, log
```

Import a subset of functionality. This can be one or more functions or variables.

Python Modules

- If we have imported an entire module, we then prefix the function names with the module name followed by a dot (i.e. dot notation).

```
import math
x = math.sqrt(9)
print(x)
```

```
3
```

If we forget to import the module...

```
x = math.sqrt(9)
```

```
NameError: name 'math' is not defined
```

- If we have imported a subset of a module, we can access all of those functions or variables without requiring the module name as a prefix.

```
from math import sqrt, log
x = sqrt(9)
y = log(2)
print(x+y)
```

```
3.6931471805599454
```

Now if we include the prefix in the call, it won't work...

```
x = math.sqrt(9)
```

```
NameError: name 'math' is not defined
```

Built-in Modules

- The Python standard library contains a large number of built-in modules for performing different tasks:

Name	Description
sys	Program control, command line arguments (e.g. exit, argv)
math	Basic mathematical functions (e.g. sqrt, exp, sin, cos)
os	File/directory operations (e.g. listdir, mkdir, rmdir)
random	Generate pseudo-random numbers (e.g. random, randint)
re	Provides regular expression matching and replacement operations

Full list of standard modules: https://docs.python.org/3/library

Basic Mathematical Functions

- Python has a `math` module that provides most basic mathematical functions. Before we can use it, we have to import it...

```
import math
x = math.sqrt(9)
print(x)
```

- We can then call various familiar functions:

```
ratio = signal_power / noise_power
decibels = 10 * math.log10(ratio)
```

```
radians = 0.6
height = math.sin(radians)
```

- We can also access variables contained in the module:

```
degrees = 45
radians = degrees / 360.0 * 2 * math.pi
answer = math.sin(radians)
```

`pi` is a variable defined in the math module

- Alternatively we could have imported the subset we required:

```
from math import pi, sin
degrees = 45
radians = degrees / 360.0 * 2 * pi
answer = sin(radians)
```

now no prefix required to access `pi` variable

Random Number Generation

- The `random` module provides functions that generate pseudorandom numbers - i.e. not truly random because they are generated by a deterministic computation, but are generally indistinguishable from them.
- The function `random()` returns a random float between 0.0 and 1.0. Each time we call it, we get the next number from a series.

```
import random
for i in range(4):
    y = random.random()
    print(y)
```

```
0.21660381103801063
0.10168268009500758
0.5845753014438958
0.443649767624016
```

Prefix functions with random. after importing!

- The module contains many other functions - e.g. `randint()` returns a random integer from the specified range.

```
import random
for i in range(4):
    y = random.randint(10,20)
    print(y)
```

```
20
14
20
15
```

e.g. return a random integer, from 10 to 20 inclusive.

Accessing Files and Directories

- The built-in `os` module provides comprehensive functionality for working with files and directories.

Get the current working directory

```
import os
print( os.getcwd() )
```

```
/home/user/Downloads
```

Prefix functions with os. after importing!

Change the current working directory

```
os.chdir("/usr/local")
print( os.getcwd() )
```

```
/home/user/Downloads
```

Get list of files in specified directory

```
os.listdir("/home/user/Documents")
```

```
['letter.doc','names.xls','results.doc']
```

Delete a file

```
os.remove("letter.doc")
```

Create a directory

```
os.mkdir("python")
```

Writing Python Scripts

- As well as IPython Notebooks, we will frequently need to write `.py` files either as stand-alone scripts or to create new modules.
- Many editors available for creating Python script files
  e.g. PyDev, PyCharm, Gedit, Sublime Text, Textmate, Notepad++
- Basic steps:

  1. Write your script in a text editor.
  2. Save your script as a `.py` file - e.g. hello.py
  3. In the terminal, change to the directory containing the script.
  4. Run python, passing the script filename as the first argument:

File: hello.py

```
for i in range(3):
    print("Hello world")
```

```
~> python hello.py
Hello world
Hello world
Hello world
~>
```

Debugging Python Scripts

- Debugging becomes a little more complicated when working with scripts. Need to be able to read output of Python error messages!
- Follow the traceback provided by Python to identify the original source of the error in the script...

File: squares.py

```
1  def add_squares( x, y ):
2     sx = x * x
3     sy = y * y
4     return sx + sy
5
6  result1 = add_squares( 3, 4 )
7  print("Result 1 = %d" % result1)
8
9  result2 = add_squares( 3, "9" )
10 print("Result 2 = %d" % result2)
```

```
~> python squares.py
Result 1 = 25
Traceback (most recent call last):
  File "squares.py", line 9, in <module>
    result2 = add_squares( 3, "9" )
  File "squares.py", line 3, in add_squares
    sy = y * y
TypeError: can't multiply sequence by non-
int of type 'str'
```

Error message is `TypeError: can't multiply sequence by non-int of type 'str'`

Error originated in `line 3, in add_squares    sy = y * y`

Command Line Arguments

- Many programs allow command-line arguments to be specified when they are run.
- A command-line argument is the information that follows a program's name on the terminal / command line, when it is executed.
- These arguments are used to pass information (e.g. file paths, options etc). to the program.

```
myprog argument1 argument2 argument3
```

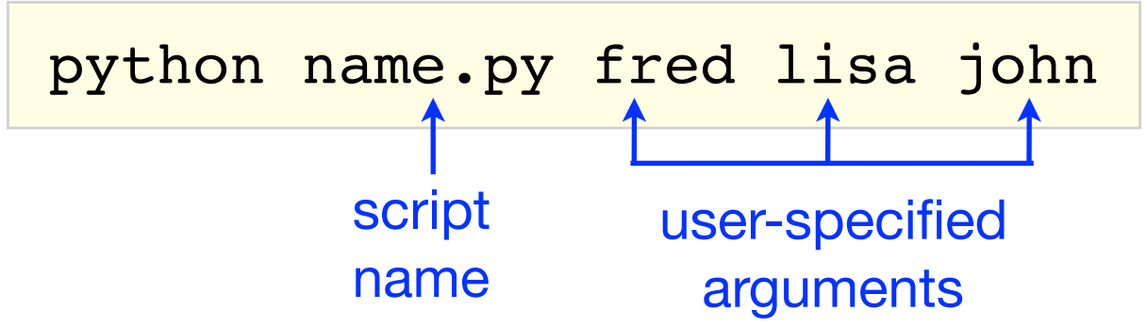
```
cd /home/alice/code
```

```
find /home/alice -type f -name README
```

```
python myscript.py
```

Command Line Arguments

- Command-line arguments are used to pass information when you start a Python script from the terminal.



- The `argv` variable in the `sys` module contains the list of command line arguments passed to the current script.

File: name.py

```
import sys

print("Got %d arguments" % len(sys.argv))
print("Script name is %s" % sys.argv[0])

for name in sys.argv[1:]:
  print("Hello %s" % name )
```

```
python name.py fred lisa john
```

```
Received 4 arguments
Script name is name.py
Received parameter fred
Received parameter lisa
Received parameter john
```

Contents of `sys.argv` above is

`['name.py', 'fred', 'lisa', 'john']`

Using Scripts as Modules

A single script file corresponds to a single module. You can import any script into another script.

File: fibo.py

```
# Build Fibonacci series up to n
def calc_fib(n):
    series = []
    a = 0
    b = 1
    while b < n:
        series.append(b)
        temp = a
        a = b
        b = temp + b
    return series
```

Exclude the .py file extension from module name when importing

File: testfib.py

```
import fibo
series = fibo.calc_fib(4)
print(series)
```