Introduction to Python scripting

~/IFT383/mod-5

Introduction

- Like the other languages covered in this course; Python is an interpreted scripting language
- However, unlike BASH, Awk and Perl; which are functional languages, Python is object-oriented
    - This object-oriented approach simplifies the process of building more complex applications.
- The Python runtime is available for a variety of Operating Systems, including Windows. Python is installed by default on many modern Linux systems

IMPORTANT VERSION DIFFERENCES!

- There are two versions of Python; 2.x and 3.x
- According to the welcome survey; ~70% of the class is using general.asu.edu
  - general has Python 2.7 and not 3.6
  - You could… get Python 3.7 running on general, but not worth the hassle
- Slides, examples, homework, labs and exams are designed to target 2.7
- I strongly encourage that you use Python 2.7 for this course
  - The skills are easily transferable - core concepts are the same
  - Reduces the need for grade appeals (grader is using General)
  - Good to have experience with older platform
  - In my experience; old software never dies in the corporate IT world
    - new software is expensive
    - ask me about statewide financial systems sometime :-)

Running Python

- Interactive mode
  - Enter Python statements into the terminal
  - Example;
    - python
    - print(“Hello World!”)
    - quit()
- As a script file
  - Create a script file with an appropriate interpreter line
  - Example: ./myScript.py
- Using the python command
  - As with Perl; provide a script file to the python command
  - Example: python myScript.py

A simple start (hello.py)

```
#!/usr/bin/python
# A simple script
print("Hello, World!")
print( \
   " This is \
a line that \
spans multiple lines" \
)
```

- Comments us the # character as in BASH, Awk and Perl
- Statements are terminated with a newline, no semicolon required
- A single statement can span multiple lines by adding a \ to escape the newline character

Identifiers

- Identifiers are the names we assign to variables, functions and other components of out program
- Rules for identifiers
  - Python is case sensitive; myVar is different than myvar
  - An identifier can contain numbers, letters and underscores “_”
  - Identifiers may not start with a digit
  - There is no limit to the length of an identifier
  - Identifiers cannot share names with Python keywords; such as if, while, import, print etc.

Common identifier conventions

- normal variables
  - Typically all lower case, or cammel-case
    - myvar, myVar
- constant variables
  - By convention; distinguished a value that is not intended to be changed
  - typically identified in all upper-case letters
  - words are either concatenated together, or separated by an “_”
    - PI, DAYS_IN_WEEK, INCH_FEET
- Class names
  - Classes are the building blocks of object-oriented languages
  - Names are in ‘title case’; the first letter is capitalized
    - SomeClass

Defining variables

- Variables must be defined before they can be used in an expression
- A variable is defined when a value is assigned to it
  - Example; coffee=10
  - This differs from Awk and Perl, where an undeclared variable in an expression would assume a default value
- An expression is anything that contains an operator
  - Example; print(myVar + 100)

Variable definition example

vars-broken.py

```
#!/usr/bin/python
myVar+=1
print(myVar)
```

This example will not run; as myVar has not been defined before it is used in an expression

vars-working.py

```
#!/usr/bin/python
myVar=0
myVar+=1
print(myVar)
```

This example works correctly

Using variables in expressions

- There are no special sigils needed to reference a variable
- variables are referenced using their identifiers

Example;

```
#!/usr/bin/python
total=12.36
tip=0.15
total += total * tip
```

Variable scope

- In Perl; we briefly discussed variable scope when using the strict pragma
- In Python; variables are limited to the scope in which they are defined
- In the following example; the variable ‘something’ only exists within its function

```
#!/usr/bin/python
myVar=0
def aFunction():
   something=100
   pass
aFunction()
print(something)
```

Interactive input

- The **input()** function can be called to prompt the user for input
  - similar to the read command in BASH
  - A string can be provided to the function, which will be used as the prompt to the person at the terminal
  - The resulting input is interpreted literally; strings must be encapsulated in quotes (Python 2.x quirk)
- The **eval()** function converts strings to an appropriate number type and can evaluate mathematical expressions stored in strings

```
#!/usr/bin/python
# Demo of the input function
myVar = input("Please enter a number: ")
myVar = eval(myVar)
print(float(myVar)/3)
```

Variable type

- Python variables have an identifier (name), value and a type
- We can check the type of a variable using the type() built-in function
- Some variable types
  - Integer
  - Float
  - String
  - List (array)
  - Dictionary (hash map)
  - File
  - Tuple (immutable list)
  - Set (unordered list)

Looking at types (types.py)

```
#!/usr/bin/python
myString="HELLO!"
myNumber=123456
myFloat=3.14159
myList=[1, 2, 3, 4]
print type(myString)
print type(myNumber)
print type(myList)
print type(myFloat)
if (type(myString) is str):
   print "We have a string!"
```

Numbers

Numbers

- There are two data types for numbers; integer and float
- Integers
  - int - integer values less than 1,000,000,000,000,000,000
  - bool - either false (0) or true (1)
  - long - values equal to or greater than 1,000,000,000,000,000,000
- Floating-point numbers
  - float - a number with a fractional portion
- Python supports the common expressions we have seen thus far
  - The data type of the result of an expression depends on its operands
    - division of two integers will result in an integer
    - division of an integer and a float will result in a float

Expressions example (expressions.py)

```
#!/usr/bin/python

print 1 + 1     # Addition
print 1 - 1     # Subtraction
print 10 * 2    # multiplication
print 1/2       # division
print 1/2.0
print 10//3.0   # integer division
print 2**8      # exponent
print 10%3      # modulus
```

Number Operators

- Python supports many of the operators we have seen in awk and python
  - With the important exception of the increment “++” and decrement “--”

```
myVar = 0;
myVar += 1 # Add and assign result
myVar -= 1 # Subtract and assign result
myVar *= 2 # multiply and assign
myVar /= 2 # divide and assign
```

```
>>> myVar=0
>>> myVar++
  File "<stdin>", line 1
    myVar++
          ^
SyntaxError: invalid syntax
>>>
```

Built-In number functions

- Type conversion
  - int(), long(), bool(), float() and complex() will convert the argument provided to the respective type
  - Example; int(“1234”) would return an integer 1234
- Mathematical operations
  - abs(n) - absolute value of n
  - divmod(n1, n2) - divides n1 and n2 and returns a tuple containing the integer division and the remainder (modulus)
  - pow(n, m) - raises n to power m
  - round(f, p) - rounds f to precision (number of decimals) p. If p is not provided; defaults to 0
    - Example: round(3.14159,2) would yield; 3.14

Built-In number functions (continued)

- Base conversions
  - hex(n) - convert an integer to hexadecimal notation
  - oct(n) - convert n to base-8 form
- ASCII conversions
  - chr(number), ord(character), unichr(number) convert to/from ASCII/Unicode
- Boolean
  - bool() - converts strings or numbers to either true or false
  - Example; bool(“”) returns false, bool(“Hi!”) returns true

Modules

- Modules contain additional functions that you can call from your Python script
- There are some that come standard with Python and many more that can be added
- Some common modules
  - sys - system functions such as reading from stdin
  - socket - interact with the network
  - math - additional mathematical functions
- Using modules
  - You import modules into your script in order to use them
  - these imports happen at the beginning of your script
  - Example
    - import math

Using sys and math modules (math.py)

```
#!/usr/bin/python
# Demo reads stdin and does math
import math
import sys

myInput = sys.stdin.readline()
myInput = eval(myInput)
print(myInput * math.pi)
```

A list of default modules and what they provide is available at;
https://docs.python.org/2/py-modindex.html

Random numbers

- The **random** module contains functions for generating random numbers
- Some of the most commonly used are;
  - random.randint(n, m) - returns a random number between n and m (inclusive)
  - random.uniform(n, m) - Returns a float between n (inclusive) and m (exclusive)
  - random.random(returns a float between 0 (inclusive) and 1 (exclusive)
  -

Sequences
Strings and Lists

Sequences

- A category of Python types that stores data in an ordered sequence
  - strings
    - A sequence of characters
  - lists
    - An array of elements
  - tuple
    - An unmodifiable list
- Accessing members of a sequence
  - Each element is indexed starting at 0
  - Multiple elements can be selected using the slice operator [n:m]
  - Sequences are similar to arrays in Awk and Perl

Sequence operators

- Membership (**in** and **not in**)
  - Tests if something is part of a sequence
    - strings - test if a single character is in a string
    - lists and tuples - test if entire element is in the sequence
- Concatenation (+)
  - Join together two sequences of the same type
    - Example;
      - “Hello, ” + “World!”
      - list1 + list2

Sequence Operators (continued)

- Repetition (*)
  - Joins copies of a sequence together
  - Useful for repeating strings or populating a list with default data
  - Example;
    - “-” * 30
      “------------------------------”
- Slices
  - Creates a new sequence from a portion of an existing sequence
  - syntax; [n:m] from index n to (but not including) index m
  - Example;
    - "0123456789"[1:4]
      ‘123’

Built-In functions for sequences

- Type conversions
  - str(object) - convert object to string
  - list(n) - converts an iterable object to a list
  - tuple(n) - convert an iterable object into a tuple
- Operational functions
  - len(n) - Returns the number of items in the sequence
  - sum(n) - Returns the sum of all items in the sequence
  - max(n) - Returns the highest element
  - min(n) - returns the lowest element
  - sorted(n) - Returns a sorted list (more parameters available)
  - zip(list1, list2, listN) - returns a tuple or elements that appear in all lists

Strings

Strings

- Strings in python are treated as a sequence of characters
- They are created when the value being assigned to a variable is in quotes
  - Unlike BASH and Awk; Python does not have a special behavior for double vs single quotes
  - To interpret a string as a literal, specify a “raw string” using r, such as;
    - “Hello!\n” - Hello with newline character
    - ‘Hello!\n’ - exactly the same as previous example
    - r”Hello!\n” - a raw string that contains a literal “\n” and no newline
- Strings are immutable; the contents of a string cannot be changed. Rather, a new string is created and takes the place of the original string

Slicing strings

- We know that slicing can be performed on any sequence. In Python, strings ate a type of sequence; allowing us to use the slice operator

```
mySting = “Stardew Valley”
print mySting[0]          # prints ‘S’
print mySting[8:]         # prints ‘Valley’
print mySting[:7]         # prints ‘Stardew’
print mySting[4:7] # prints ‘dew’
```

String operators

- Strings inherit all operators common to sequences; concatenation repetition etc.
- String-specific operators
  - Format string (%)
    - similar to printf in other languages
    - “format string” % (argument list)
      - %c - character
      - %s - string
      - %i - integer
      - %f - float

String Format example (formatter.py)

```
#!/usr/bin/python
myChar = "?"
myString = "Chelsey"
myInt = 100
myFloat = 36.52

# example using concatination
print( "Hello, " + myString + "! You have " + str(myInt) + " new\
messages, and $" + str(myFloat) + " in your bank account" + myChar)

# Example using formatter
print ("Hello, %s, You have %i new messages, and $%.2f in your bank account%c") \
   % (myString, myInt, myFloat, myChar)
```

String functions

- Strings can use all of the functions available to sequences
- Additionally, there are a few string-specific functions
  - cmp(string1, string2)
    - Compares the ASCII values of two strings and returns
      - 0 - strings are equivilant
      - -1 - String 1 is a lower ASCII value than string 2
      - 1 - String 2 has a lower ASCII value than string 1
    - ASCII defines the integer values in memory that represent characters

string functions (string module)

- Importing the **string** module gives us access to even more string-specific functions!
- complete list at; https://docs.python.org/2/library/string.html#module-string
  - split(string, seperator)
    - Splits the provided string using the separator and returns an array
  - find(string, substring, [start], [end])
    - Searches **string** for **substring** and returns the index of **substring**
    - Optionally, start and end control where the search is performed
  - replace(string, old, new, [limit])
    - Searches **string** for **old** and replaces it with **new**
    - Optionally, limit restricts the number of replacements
  - join(list, seperator)
    - Join elements of **list** using **seperator**

NOTE: In Python 3, most of these are now part of the str class

string module example
practicallyAwk.py

```
#!/usr/bin/python
from string import replace
from string import split
inputString = "First,Second,Third,Fourth\n"

# remove the newline
inputString = replace(inputString, "\n", "")
fields = split(inputString, ",")

# print the first few fields
print(fields[0])
print(fields[1])
```

Lists

List - like an array, but better

- Provides an indexed, ordered collection of elements in one variable
- Unlike strings; can contain any data type, even lists!
- Lists are flexible
  - Add, remove, combine, sort and split elements of a list
- Creating a list
  - myList = [1, “2”, 3] # predefined list
  - myList = list() # empty list
- Access
  - myList[0] # 0 is the index and counts up for each element
- Remove elements
  - del myList[1] # removes the element at index 1
  - myList.remove(“Blue”) # removes the element equal to “Blue”

Lists in lists

- Lists can contain lists
  - Similar to multi-dimensional arrays
- Creation
  - fourByFour = [[“A1”,”A2”],[“B1”,”B2”]]
- Access
  - fourByFour[0][0]
    returns: “A1”

List functions

- list.append(object)
  - add **object** to the end of the list
- list.index(object)
  - Returns the position in the list that contains **object**
- list.pop()
  - return the object at the end of the list and remove it
- list.reverse()
  - Reverse the elements in the list
- list.sort()
  - Sort the elements in the list
- list.remove(object)
  - Remove **object** from the list

List example (grocery.py)

```
#!/usr/bin/python
# Demo of list functions
from string import join

gList = list()
print( len(gList) )
gList.append("Bread")
gList.append("Dog food")
gList.append("Eggs")
print( "there are %i items in the list!" % (len(gList)))
gList.remove("Bread")
print( "removed bread; there are now %i items in the list!" % (len(gList)))
print( join(gList, "\n") )
```

Dictionaties
(fancy hash maps)

Introduction

- Dictionary objects correlate keys (also called hashes) to values
  - Similar construct to hash variables in Awk and Perl, but object-oriented
- Data is stored and accessed differently than the structures we have seen thus far
  - Sequence types store data indexed by an integer starting at 0
  - Dictionary data types use objects to index values
    - strings are the most common
  - Providing a key to a dictionary maps directly to the associated value

Creation and Assignment

- Create a dictionary
  - myDictionary = {} # Creates an empty dictionary
  - myDictionary = {“key1”: “value1”, “key2”: “value2”} # new dictionary with data
  - myDictionary = dict([“key”,”value”],[“key”,”value”])
- Creating with default values
  - myGrades = {}.fromKeys((“Homework”,”Lab”,”Quiz”), 0)
    - Creates a dictionary with the keys; Homework, Lab and Quiz
    - Sets the associated value of those keys to 0
    - Recall that you cannot use an undefined variable in an expression as we did in Awk and Perl; this provides an easy method to define some default values

Dictionary access

- Access specific value
  - use square brackets and the key
    - print (myDictionary[“key1”])
- List all keys
  - Obtain a list of all keys in the dictionary
    - myDictionary.keys()
- Check if key exists
  - Use the ‘in’ and ‘not in’ keywords
    - ‘key1’ in myDictionary

Modifying a dictionary

- Add a new key/value pair
  - Use the new key as part of an assignment
    - myDictionary[‘key1’] = ‘Some value’
- Modify an existing value
  - Modifying existing values is exactly the same procedure as creating new pairs
    - myDictionary[‘key1’] = ‘a new value’
  - Once a pair exists, you can use it as part of an expression
    - myDictionary[‘key1’]+=1
- Removing a key/value pair
  - del myDictionary[‘myKey’] # remove myKey and its value
  - myDictionary.pop[‘myKey’] # return value and delete ‘myKey’

Type operators and functions

- Dictionaries do not support concatenation (+) or repetition (*)
- len(dictionary)
  - Returns the number of key/value pairs in the dictionary
  - example;

```
#!/usr/bin/python
# dictionary length example
childAges = {"Andrew": 6, "Hannah": 3, "Ava": 1}
print( len(childAges) )
```

output: 3

Dictionary example (acronyms.py)

```
#!/usr/bin/python
import string

# Define a small dictionary
acronyms = {
    "UTO":"University Technology Office",
    "EAS":"Engagement and Advising Services",
    "TPS":"The Polytechnic School",
    "GNU":"GNU is Not UNIX",
    "OSD":"Operating System Deployment"
}

# Display a list of keys and ask the user which one they would like to see
keyList = string.join(acronyms.keys(), "\n")
print (keyList)
selection = input("Please enter one of the entries above to see definition")

# Convert to uppercase letters
selection = string.upper(selection)

# Generate and print key and associated value
outputString = "\n%s\n\t%s\n" % (selection,acronyms[selection])
print(outputString)
```